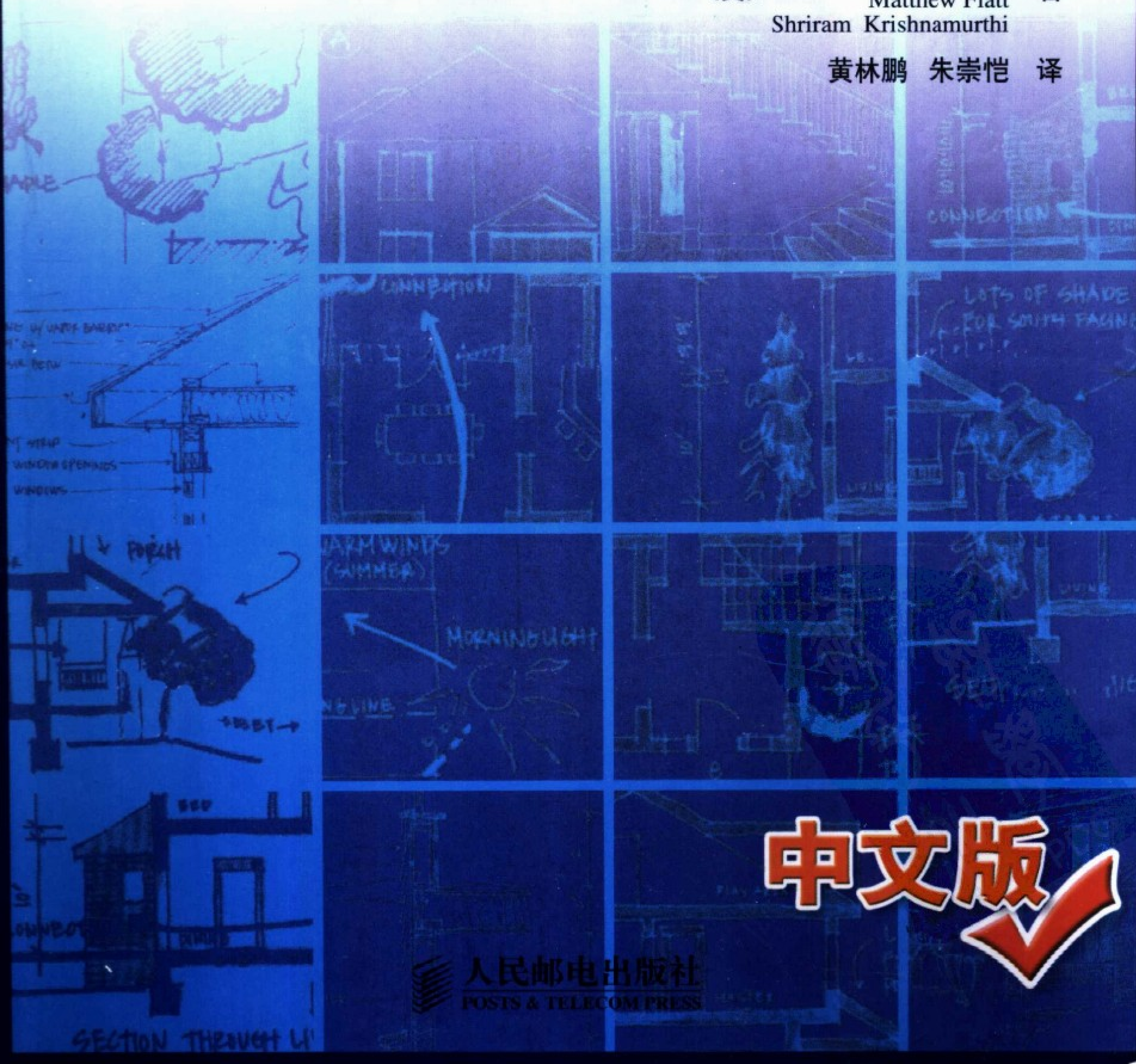
国外著名高等院校
信息科学与技术优秀教材

程序设计方法

How To Design Programs

〔美〕 Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi 著

黄林鹏 朱崇恺 译



中文版

人民邮电出版社
POSTS & TELECOM PRESS

国外著名高等院校信息科学与技术优秀教材

程序设计方法

Matthias Felleisen

［美］ Robert Bruce Findler 著

Matthew Flatt

Shriram Krishnamurthi

黄林鹏 朱崇恺 译

人民邮电出版社

图书在版编目（CIP）数据

程序设计方法 /（美）傅雷森（Felleisen,M.）等著；黄林鹏等译.
—北京：人民邮电出版社，2003.12
ISBN 7－115－11556－7

Ⅰ. 程… Ⅱ. ①傅… ②黄… Ⅲ. 程序设计—方法 Ⅳ. TP311.11

中国版本图书馆 CIP 数据核字（2003）第 074636 号

版 权 声 明

国外著名高等院校信息科学与技术优秀教材

程序设计方法

◆ 著 [美]Matthias Felleisen Robert Bruce Findler Matthew Flatt Shriram Krishnamurthi
译 黄林鹏 朱崇恺
责任编辑 陈冀康

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www. ptpress.com.cn
读者热线 010-67132705
北京汉魂图文设计有限公司制作
北京隆昌伟业印刷有限公司印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：30
字数：925 千字 2003 年 12 月第 1 版
印数：1-4 000 册 2003 年 12 月北京第 1 次印刷

著作权合同登记 图字：01－2002－1069 号

ISBN 7-115-11556-7/TP・3578

定价： 92元

本书如有印装质量问题，请与本社联系 电话：(010)67129223

内容提要

本书以 Scheme 语言为基础介绍计算和程序设计的一般理论和实践。

本书由 8 个部分和 7 个独立的章节（第 8、13、18、24、29、33、38 章）组成。8 个部分主要讨论程序设计，独立章节则介绍一些与程序设计和计算相关的话题。本书第 1 至第 3 部分介绍了基于数据驱动的程序设计基础。第 4 部分介绍了程序设计中的抽象问题。第 5 部分和第 6 部分是与递归及累积相关的内容。本书的最后两部分说明了设计程序的意义，阐述了如何应用前 6 个部分所描述的程序设计诀窍，以及使用赋值语句必须特别小心的一些问题。

本书可作为高等院校计算机科学与技术专业“程序设计导论”和“计算导论”的教材和教学参考书，也可作为函数式语言和 Scheme 语言的入门教材。



前　言

向儿童传授程序设计知识有悖于现代教育学。制定计划、学习教规、注重细节、严格自律有何乐趣？

——艾伦·佩利（1966年图灵奖获得者），《编程警句》

许多职业都需要进行某种形式的计算机编程。会计师使用电子表格和字处理软件编程，摄影师使用图片编辑器编程，音乐家使用音响合成器编程，职业程序员使用计算机编程。编程已成为一种人人都需要掌握的技能。

编写程序并不仅仅是一种职业技能。事实上，好的编程是件有趣的事，是一种创造性的情感发泄，也是一种用有形的方式表达抽象思维的方法。程序设计可以教会人们多种技能，如阅读判断、分析思考、综合创造以及关注细节，等等，这些技能对各种类型的职业来说都是重要的。

所以，在普通教育中，程序设计课程的地位应该和数学、语文一样重要。或者用更简洁的话来说，就是

每个人都应该学习如何设计程序。

一方面，程序设计跟数学一样，可以训练人的分析能力，不同的是，程序设计是一种积极的学习方法。在与软件的互动过程中，学生可以直接得到反馈，进行探索、实验和自我评价。与钻研数学习题相比，程序设计的成果，即计算机软件，更有趣，也更有用，它们能极大地增加学生的成就感。另一方面，程序设计跟语文一样，可以增强学生的阅读和写作能力。即使是最小的编程任务，也是以文字形式表达的，没有良好的判断和阅读技能不可能设计出符合规范的程序，反之，好的程序设计方法会迫使学生用适当的语言清晰地表达他的思考过程。

本书是基本的程序设计教科书，讨论如何从问题描述产生组织严谨的程序。本书把注意力集中于程序的设计过程，不强调算法和语言细节，不注重于某个特定的应用领域。这门介绍性的程序设计课程有两个根本性的创新。创新之一是给出一系列明确的程序设计指导。现有的程序设计课程往往趋向于给出含糊的、不明确的建议，如“自上而下设计”或者“结构化程序设计”等。与此不同，本书给出了一系列程序设计指导，由此引导学生一步一步地从问题的描述出发，通过明确定义的中间过程，得出程序。在这个过程中，学生将学会阅读、分析、组织、实验和系统思维能力。创新之二是使用了一个全新的程序设计环境。过去的编程教材往往简单地假设学生有能力使用某种专业程序开发环境，而忽略程序设计环境对学生学习的影响。本书为初学者提供的程序设计环境会随着你所掌握的知识的多少而改进，该环境最终可

以支持完整的 Scheme 语言，使用该语言既可以编写大型程序又能编写脚本程序，可以完成所有领域的编程任务。

本书讨论的编程指导以程序设计诀窍（programming design recipe）阐述[1]。设计诀窍指导程序设计初学者逐步掌握问题求解的过程。有了设计诀窍，程序设计的初学者就不用再盯着空白的纸张或计算机屏幕发呆了，他们可以自我检查并核对设计诀窍，使用“问答”方式进行程序设计并取得进步。

本书通过识别问题的范畴来建立设计诀窍，而问题范畴的识别基于表示相关信息的数据类型。从该数据类型所描述的结构出发，你可以用一个清单推导出程序。图 1 给出的设计诀窍包含了程序设计的 6 个基本步骤，每个步骤都将产生定义明确的中间结果：

1. 问题数据类型描述；
2. 程序行为的非正式描述；
3. 说明程序行为的例子；
4. 开发程序的模板或视图；
5. 把模板转换成完整的定义；
6. 通过测试发现错误。

程序设计决窍的基本部分
1. 问题分析和数据定义
2. 合约，用途说明与结果的描述，函数头部
3. 例子
4. 函数模板
5. 函数定义
6. 测试

图 1　程序设计决窍的基本步骤

主要差异在于第 1 步和第 4 步之间的关系。

使用设计决窍不仅对初学者有所帮助，对教师也有益。教师可以使用清单检查初学者解决问题的能力，诊断错误所在，并提出具体的纠正措施。毕竟，设计决窍的每一阶段都会产生一个定义明确、可检查的结果。如果一个初学者遇到了困难，教师可以借助清单检查他的中间结果，并判断问题之所在。教师还可以针对程序设计决窍中某一特定的过程给学生提供指导，提出合适的问题，并推荐额外的练习题。

为什么每个人都应该学习编写程序

想象会把不知名的事物用一种形式呈现出来，诗人的笔会使它们具有如实的形象，空虚的事物也会有了居处和名字。

—— 莎士比亚，《仲夏夜之梦 V(i)》

目前越来越少的人在编写程序代码，主张每个人都应该学习编程似乎有些奇怪。事实上大多数人是在使用应用程序包开发软件，即使是程序员也使用“程序生成器”由规则（如商业规律）创建程序，看起来他们似乎不需要编写代码。那么，为什么还说每个人都应该学习编程呢？

问题的答案可以从两个方面阐述。第一，传统形式的编程确实仅仅对少数人来说是有用的。但我们这里所讨论的编程模式对每个人，不管是使用电子表格的行政办公室秘书还是高科技公司的程序员，都是有用的。换句话说，这里所讨论的编程概念远比传统的编程观念广泛。第二，本书以最小影响原则来

[1] 那些熟悉 C/C++、Basic 和 Pascal 等程序设计语言的读者可以将前言中提到的程序（program）理解为过程（procedure）或方法（method）。

讲授编程思想，着重于分析问题和解决问题的技能，而不是强迫大家掌握传统的编程语言和编程工具。

要想更好地理解现代编程思想，请仔细观察一下目前流行的应用程序包，如电子表格。如果用户先把描述一个单元 A 和另一个单元 B 依赖关系的公式输入电子表格，接着，输入单元 B 的值，电子表格就会自动计算单元 A 的值。对于复杂的电子表格，一个单元的值可能依赖多个其他单元，而不仅仅是一个。

其他应用程序包也需要类似的计算。考虑文字处理和样式表软件。样式表说明了如何由待定的词或句建立一个（或一部分）文档。当提供了特定的词句之后，文字处理软件就会把样式表中的名字替换为特定的词句，从而建立文档。类似地，某个进行网页检索的人可能会指定若干个关键字、给定关键字之间的顺序以及哪个关键字不必在网页中出现。在这种情况下，搜索结果将取决于搜索引擎的高速缓存和用户所输入的检索表达式。

最后，使用程序生成器的技巧其实就是使用应用程序包的技巧。程序生成器由高层功能描述生成传统程序设计语言代码，例如由商业规律或科学定律产生 C++或 Java 程序。规律把数量、销量以及库存记录联系起来并说明计算过程。而程序的其余部分，特别是如何与用户交互以及如何将数据存储于计算机磁盘等等，则几乎或完全不需要人的干预。

所有这些活动都是让计算机软件为我们做某些事。其中一些活动使用科学符号，一些使用固定格式的自然语言，另一些则使用具体的编程符号。实际上这些活动都是某种形式的编程，其本质可归结如下：

1. 把某个量与另一个量相关联；
2. 用值代换名进行关系计算。

事实上，上述两个概念刻划了使用最低级的程序设计语言，如机器语言，和使用最流行的程序设计语言，如 Java，进行编程的本质。程序将输出和输入相联系，将程序应用于特定的输入，就是在计算中用具体的值代替相关的名字。

没有人可以预知今后 5 年或是 10 年内会出现哪种类型的应用程序包。但是，使用应用程序包仍然需要某种形式的编程。要使学生们掌握编程，学校要么强迫他们学习代数，它是编程的数学基础，要么让他们进行某种形式的程序设计活动。有了现代化的程序设计语言和程序设计环境，选择后者可以更有效地完成任务，还可以使代数学习的过程变得更加有趣。

设计诀窍

烹饪既是孩童的游戏也是成人的乐事，细心烹饪是爱的举措。

——克雷格·卡莱波恩（1920—2000），《纽约时报》饮食版编辑

学习设计程序就像学习踢球一样，必须练习断球、运球、传球和射门。一旦掌握了这些基本技术，下一个目标就是学习担任某个角色、选择并实施合适的战略，如果没有现成的，需要创造一种。

程序员和建筑师、作曲家以及作家一样，是富有创造性的人。他们的念头从白纸开始，先构思概括，再把它写到纸上，直到写出的东西能充分反映他们的思想为止。他们使用图形、文字或其他方法来表达建筑物风格、描述人的特征或是谱写音乐旋律。他们能胜任自己的职业，是因为经过长时间的练习，他们能本能地使用这些技能。

程序设计者也是先形成程序框架，然后翻译为最初的程序版本，再反复修改，直到与最初的想法相符。事实上，好的程序员会多次编辑和修改自己的程序，最终达到某种形式的标准。这和足球运动员、建筑师、作曲家以及作家一样，他们必须长期练习行业必需的基本技能。

设计诀窍类似于控球技巧、写作技巧、乐曲编排技巧和绘图技巧。通过学习和研究，在程序设计领域，计算机科学家已经积累了许多重要的方法和技巧，本书挑选了其中最重要和最实用的一些，由浅入深，逐一讲解[1]。

[1] 我们的设计诀窍参考了 Daniel P. Friedman 关于结构递归的工作、Robert Harper 关于类型理论的工作以及 Michael A. Jackson 关于设计工作的方法。

本书大约有一半的设计诀窍涉及输入数据和程序之间的关系。更准确地说，它们描述了如何从输入数据的描述得出整个程序的模板，这种基于数据驱动的程序设计方式最常见，易于创建、理解、扩展和修改。其他设计诀窍有生成递归（generative recursion）、累积(accumulation)和历史敏感性(history sensitivity)。其中，递归型程序可以被重复调用以处理新的问题；带累积器的程序在处理输入的过程中收集数据；历史敏感性程序可以记住程序被多次调用的信息。最后，但不是最不重要的，是抽象程序的设计诀窍。抽象是把两个（或更多）相似的设计概括为一个并由它衍生最初示例。

在许多场合下，往往会由问题联想到设计诀窍。在另外一些场合下，则必须在几种可能性中作出选择，不同的设计决窍可能会导致不同的程序结构，它们之间的差别可能很大。对于一个具有创造性的程序员来说，做出选择是很自然的事情。除非程序员十分熟悉所有可选的设计诀窍，完全理解选择某个诀窍而不是另一个诀窍的后果，否则程序设计过程不可避免按事论事，甚至会导致离奇古怪的结果。我们希望通过制订一系列设计诀窍来帮助程序员理解选择什么以及如何进行选择。

上面解释了“编程”和“程序设计”的含义，读者应该理解到本书所讲授的思想方法和技能对多种职业来说都相当重要。要正确地设计程序，你必须：

1. 分析通常使用文字表述的问题；
2. 在抽象表达问题实质的同时使用例子进行说明；
3. 用精确的语言阐明所表述的语句和注释；
4. 通过检查、测试对上述活动进行评价和修改；
5. 关注细节。

所有这些行为对商人、律师、记者、科学家、工程师以及其他人来说都是有用的。

尽管传统意义上的编程也需要这些技巧，但初学者往往不理解它们之间的关系。问题是，传统的程序设计语言以及传统形式的编程需要学生完成大量的登记工作并记住许多与特定语言相关的细节。简而言之，琐碎杂事淹没了技术本质。要避免这个问题，教师必须使用一种适合初学者的程序设计环境，它尽可能不增加学生额外的负担。在开始编写本书的时候，这样的工具并不存在，因此我们就自行开发了。

选用 Scheme 和 DrScheme

我们把美归于简单，
不含多余部分，
边界清晰，
与一切相关联，
是中庸之道。
——拉尔夫·沃尔多·爱默生，《人生苦旅》

本书选择 Scheme 作为编程语言，辅助程序设计环境为 DrScheme，软件可以免费从本书的正式网站下载[1]。

尽管如此，本书并不是一本介绍 Scheme 程序设计语言的书籍，它仅涉及部分 Scheme 结构。具体来说，本书仅使用 6 种 Scheme 结构（它们是函数定义和调用、条件表达式、结构体定义，局部定义以及赋值等）以及大约 12 个基本函数，它们就是讲授计算和编程原则所需要的全部东西。希望把 Scheme 当作一种工具来使用的人则需要阅读其他的材料。

对初学者来说，选用 Scheme 是很自然的。首先，程序员可以把注意力集中于两个要素，即前面所指出的基本编程原则：程序就是数量之间的关系，对于特定的输入求取结果。使用 Scheme 语言核心，在教师的指导下，学生在第一堂课就可以开发出完整的程序。

[1] Scheme 有一个正式的定义，即由 Richard Kelsey、Willism Clinger、Jonathan Rees 和许多 Scheme 实现者编辑的“Scheme 修改报告”。要了解该报告以及 Schme 的不同实现，请访问 www.schemers.org。请注意，本书对该报告进行了扩充并针对初学者进行了剪裁。

其次，可以方便地将 Scheme 组织成从简单到复杂的一系列不同级别的语言。这个性质对初学者来说是至关重要的。当初学者犯了简单的符号错误时，一般程序设计语言会给出含糊的、与语言高级特征相关的错误消息。初学者往往浪费很多时间来查找错误所在，由此产生学习上的挫折感。为了避免这些问题，通过持续对赖斯大学计算机实验室的程序设计初学者的观察，经过谨慎选择，DrScheme 实现了若干不同层次的 Scheme 程序设计环境。按照安排，不同的环境会给出与学生当前知识水平相适应的错误消息。更好的是，分层会避免许多基本错误。当学生学习了足够的编程和语言知识后，教师可以建议他们接触更丰富的语言层次，由此编写更有趣、更简练的程序。

另外，DrScheme 提供了一个真正的交互式环境。环境由两个窗口组成：一个是 Definitions 窗口，在其中可以定义程序，另一个是交互窗口，其行为就像是一个袖珍计算器，你可以在其中输入表达式，由 DrScheme 求出它们的值。换句话说，计算由袖珍计算器完成，而这是学生们都相当熟悉的。很快，计算形式就从袖珍计算器上的算术运算向前推进，变成对结构体、表和树的计算。使用交互式的计算方式甚至可以鼓励学生们用各种方法进行程序实验，从而激发他们的好奇心。

最后，使用包含丰富数据结构的交互式程序设计环境可以让学生把注意力集中于问题的解决和程序设计活动之上。关键的改进是，交互式求值环境避免了（几乎是）多余的关于输入和输出的讨论。这一点改进带来了几种结果。第一，掌握输入和输出函数需要记忆，学习这些东西单调乏味、令人厌烦，相反，如果使用固定方式的输入和输出，我们就能将精力集中于问题求解技术的学习；第二，良好的面向文字的输入需要深奥的编程技能，最好从问题求解的课程中学习（学生应该从更高级的课程中学习）。教那些糟糕的面向文字的输入，是对老师和学生时间的浪费；第三，现代软件一般采用图形用户界面（GUI），GUI 是程序员使用编辑器和“向导”设计的，不是手工完成的。学生最好学习使用与标尺、按钮、文本框等相关的函数，而不是背诵那些与流行的 GUI 库相关的特定协议。简而言之，在初次介绍编程时就讨论输入和输出是对宝贵的学习时间的浪费。如果要进一步学习，掌握必需的 Scheme 输入输出知识也比较方便。

总而言之，只要少量几节课，学生们就可以学会 Scheme 语言的核心，这种语言和传统的程序设计语言一样强大。这样，学生立即就可以把注意力集中于编程本质，这将极大增强他们解决一般问题的能力。

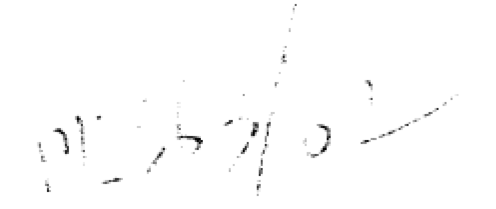
本书正文部分

本书由 8 个部分和 7 个独立的章节（书中第 8、13、18、24、29、33、38 章）组成。8 个部分主要讨论程序设计，独立章节则介绍一些与程序设计和计算相关的话题。图 2 给出了本书各部分之间的依赖关系，可以看出，你可以按不同的顺序来阅读本书，只阅读部分内容也是可以的。

本书第一至第三部分包括了基于数据驱动的程序设计基础。第四部分介绍了程序设计中的抽象问题。第五部分和第六部分与递归及累积相关。本书前 6 部分使用了纯函数式（或称代数式）的程序设计风格，即无论计算多少遍，同一个表达式每次计算的结果总是相同。这种特性使程序易于设计，程序性质易于推导。不过，为了处理程序之间的接口和解决其他领域的问题，我们放弃了部分代数性质，引入了赋值语句。本书的最后两部分说明了设计程序的意义，更精确地说，它们阐述了如何应用前 6 个部分所描述的程序设计诀窍，以及使用赋值语句必须特别小心的一些问题。

独立章节则介绍一般性的、非本质的，对计算和程序本身来说是重要的话题，但并不涉及程序设计有的是在严格的基础上介绍本书所选定的 Scheme 子集的语法和语义，有的是介绍了另外的程序设计结构。独立章节 5（第 29 章）讨论了抽象的计算开销（包括时间和空间开销），并介绍了向量的概念。独立章节 6（第 33 章）则比较了两种数值表示技术以及处理它们的方法。

只有某些独立章节的内容的学习可以推后，直到需要时再学习。对于学习与 Scheme 语法和语义相关的独立章节尤其应该注意。但是，考虑到图 2 中第 18 章的重要地位，我们应该及时学习。



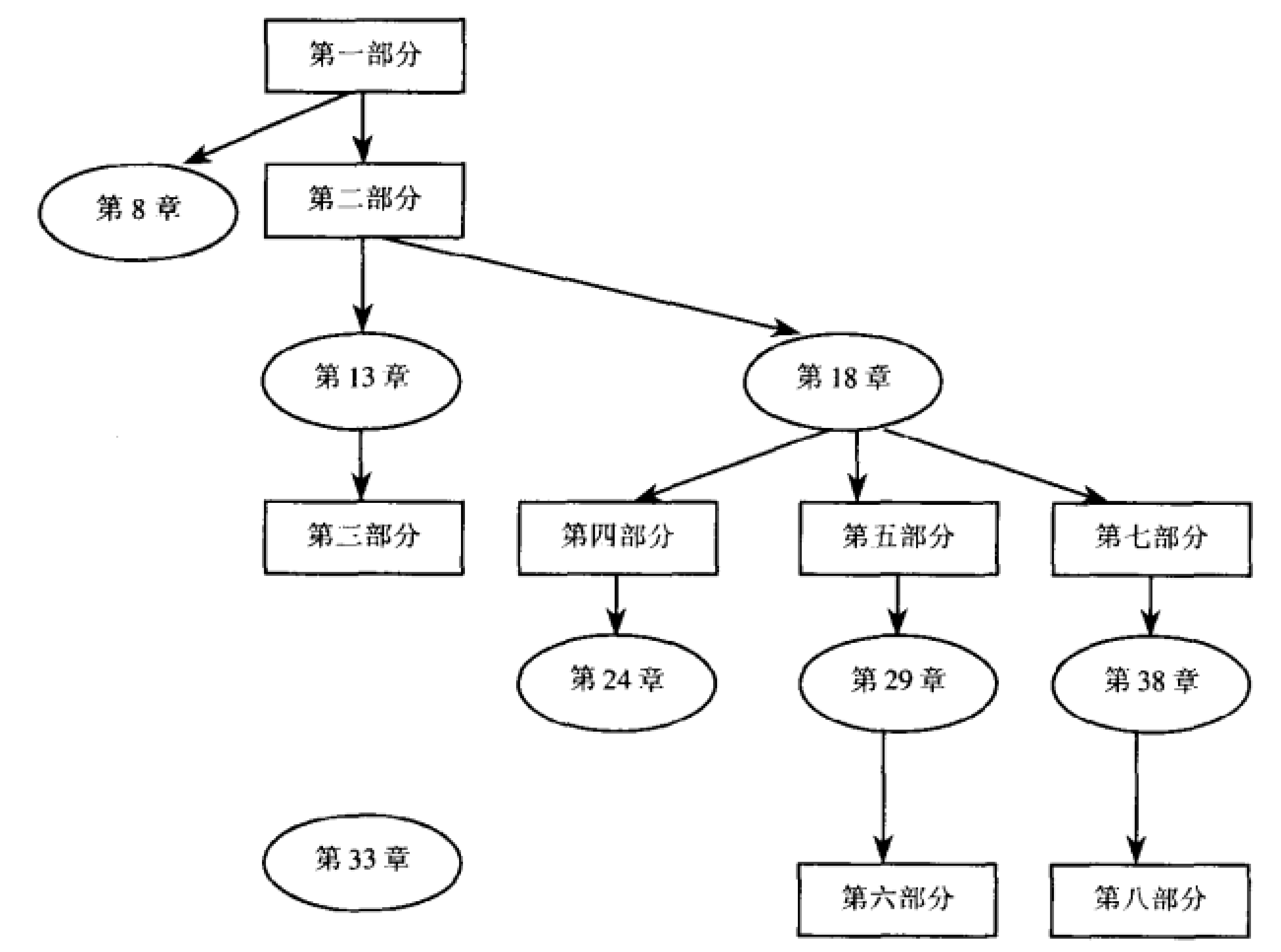


图2 本书各部分和独立章节之间的依赖关系

程序的逐步求精：系统化程序设计方法对于开发大型项目特别有意义，也特别重要。而开发单一函数到小规模的包括多个函数的项目则需要另外一种设计思想：逐步求精，即先设计程序核心，再增加功能，直至满足整个需求目标为止。

学习完第一节课后同学就应该有了程序逐步求精的初步印象。为了使学生熟悉这种技术，书中给出了许多补充练习，这些使用简短概述引出的练习可以指导学生进行程序逐步求精的训练。第16章将明确阐述这种思想。

另外，本书会重复使用某些练习和例子。例如，第6.6节、第7.4节、第10.3节、第21.4节、第41.4节以及最后2节中的一些练习题都涉及了如何在画布上移动图片的问题。这样，学生就会多次看到同样的问题，而每一次讨论都会增加他们对程序组织的了解。

本书通过逐步把功能加到一个程序体中的方法来示范为什么程序员必须遵循设计诀窍，借助问题的解决方式向学生展示如何在可用的设计诀窍中进行选择。有时候新知识的作用只是帮助学生改进程序的组织结构，换句话说，要让学生了解到在他们初步的工作完成之后，编程过程并没有结束，如撰写论文和书籍一样，程序也需要进一步的编辑和修改。

教学软件包（TeachPack）：完成工程项目的一个要求是程序员必须进行团队合作。在程序设计教学环境下，这意味着一个学生写的程序必须与另一个学生编写的程序相匹配。为了模仿“与另一个程序相配”这一概念，本书提供了DrScheme教学软件包。粗略地说，教学软件包模仿了一个合作者，由此可以避免由于合作者程序存在错误而带来的不便。技术性的说法是，工程总是由视图和程序组件模型两个部分组成（在模型—视图软件体系结构意义上），在典型的环境下，学生设计模型，教学包则提供视图。通常，教学包以（图形）用户界面的形式提供视图，而不是单调乏味、毫无意义的代码。事实上，这种分离模仿的是真实世界中的工程分工。

为了使模型与视图相符，学生必须注意函数的规范，必须遵循设计诀窍。对程序员来说，使模型与视图相符是一种非常重要的技能，但程序设计的初级课程常常未能给予足够的重视。在第四部分，为了说明创建GUI的过程并不神秘，我们将说明如何建立一些简单的GUI，以及GUI事件是如何触发函数调用的，但我们不会把很多时间花费在一个需要死记硬背，只需很少思考的主题上。

进度：根据需要，每个学校都可以有自己的教学进度表。在赖斯大学，讲授整本教材以及其他一些附加材料通常需要一个学期的时间。一个研究性大学的教师可以使用类似的进度。而高中教师就必须放

慢进度。许多尝试使用本教材的高中教师在一个学期内完成了前三个部分的教学；而少数高中只使用本书第一部分，从计算的角度讲授代数问题的求解；另一些高中则用一年的时间教完整本书。要得到有关教学进度表的更多信息，请访问本书的网站。

本书站点：本书有两种版本，除了纸介版本，在网站

http://www.htdp.org/

可以免费获得电子版本。

网站提供了一些附加材料，包括前面提到过的各种类型的补充练习。目前网站提供有可视化的小球游戏模拟，更多的练习将在不久的将来加入网站。

致谢

我们特别感谢四个人：罗伯特・科克・卡特赖特，他与本书的第一个作者合作开发了赖斯大学此入门性课程的前身；丹尼尔・P・弗里德曼，他在 1984 年要求本书的第一个作者重写了 *The Little LISPer*（由麻省理工学院出版社出版），而这也是本书写作计划的开始；约翰・克莱门特，他负责设计、实现和维护 DrScheme 软件；还有保罗・斯特克勒，他忠实地支持我们，帮助我们开发所需的程序设计工具组件。

有许多友人和同事帮助我们开发这本教材，他们在自己的课堂上使用本教材，并且对本书的初稿给出具体的评论。我们对他们的帮助和耐心表示感谢，这些人包括伊恩・巴兰德，约翰・克莱门特，布鲁斯・迪拜，迈克・厄恩斯特，凯瑟・菲尔勒，丹尼尔・P・弗里德曼，约翰・格林纳，约翰・斯通，杰拉尔丁・莫林和瓦尔迪马・武门。

在赖斯大学，本书草稿在课程 Comp 210 上使用了 12 次，学生们提出了许多宝贵意见。众多 TeachScheme！研讨会的参加者在他们的课堂上使用了本书的最初草稿，他们中的许多人提出了评论和建议。作为其中代表，这里列出一些作出过积极贡献的人，他们是：巴巴拉・阿德勒女士，斯蒂芬・布洛赫博士，杰克・克莱先生，理查德・克莱门斯博士，凯尔・吉列先生，克伦・布拉斯女士，马文・赫男得先生，迈克尔・亨特先生，克伦・诺斯女士，贾明・雷蒙德先生以及罗伯特・里德。克里斯托弗・费雷荪和他的父亲耐心地参与了本书前几个部分的工作，让我们直接了解到了年轻学生的观点。感谢他们中的每一位。

最后，Matthias 在这里表达他对海尔格的感激，感谢她多年以来的耐心，感谢她为一个心不在焉的丈夫和父亲建立了一个家庭。Robby 要感谢黄馨慧的支持和鼓励，没有她，他不可能完成任何事。Matthew 感谢顾文沅给予的忠诚的支持和不停的争论。Shriram 感激凯瑟・菲尔勒的支持、忍耐和俏皮语，同时对她参与本书部分工作表示感谢。



目录

第一部分　简单数据的处理

第二部分 任意数目数据的处理

第三部分　再论任意大数据的处理

第四部分　抽 象 设 计

第五部分 生 成 递 归

第六部分 知 识 累 积

第七部分 改变变量的状态

第八部分 复合值的改变

第一部分

简单数据的处理





第1章 学生、教师和计算机

我们从孩童就开始学习计算，最初仅仅是数的加减，如：

1加1等于2，5减2等于3。

随着年龄的增加，我们学到了更多的数值运算，如指数和三角函数等，也学到了如何描述计算规则，如：

给定一个半径为r的圆，它的周长是r乘以2π。一个劳动者（假定每小时最低报酬是5.35美元）工作N小时所获得的最低报酬是N乘以5.35美元。可以说是教师让我们成为能执行简单程序的计算机。

因此，计算本身并没有秘密可言。计算机仅仅是计算速度非常之快的学生。当我们还在思考第一道题目的时候计算机却可能已经完成了成千上百万次运算。但是，计算机程序所能做的事远远超出了数值计算得多，它可以给飞机导航，可以作为一个游戏的参与者，可以查询一个人的电话号码，也可以打印一个大公司的工资单。简而言之，计算机可以处理所有类型的信息。

人类可以使用自然语言描述信息和指令，如：

当前温度是35℃，请将其转换为华氏温度值。引擎花了35秒将汽车速度从零加速到100英里，请确定在第20秒时汽车的速度。

而计算机，基本不懂自然语言，也无法理解以自然语言表示的复杂指令。因此，为了向计算机传达信息和指令，我们不得不学习一种计算机语言。

表示指令和信息的计算机语言就是程序设计语言。以程序设计语言表示的信息称为数据。有许多种类型的数据，例如数是一种数据，而数列也是一种数据，但后者属于复合数据，因为每个数列都由较小单位的数据（即数）组成。为了区分这两种类型的数据，前者称为原子数据。字母是另一种类型的原子数据，而家谱树则是一种复合数据。

虽然数据表示信息，但它们的具体解释则依赖于我们，如数37.51可能表示一个温度，也可能表示时间或距离。而字符“A”可以表示学习成绩、食品的质量或一个地址的一部分。

与数据一样，指令（也称为操作）也有不同的风格。每类数据都与一个基本操作集相关联。例如数，相关的操作有+、-、*等。程序设计者将基本操作组合成程序。因此，可以将基本操作想象为某一外国语言中的单词，而将程序设计想象为在这种语言中遣词造句。

一些程序如同散文般简短，而另一些却如同百科全书般殷实。撰写散文和书籍需要细心的策划，编写程序也是如此。不管大小，一个好的程序不可能是修修补补的结果，它必须经过精心设计，每一部分都需要关注，要将简单程序组成一个更大的单位还必须遵循预定的规则。因此设计好的程序必须从程序设计的最初实践开始。

在本书中，我们将学习如何设计计算机程序以及理解它们的功能。成为一个程序设计者是有趣的，但不是一件容易的事。程序员人生中最好的部分是看着我们的“产品”逐渐长大并获得成功。看到自己设计的计算机程序能够玩游戏很有趣吧；看到自己设计的计算机程序能帮助他人，很兴奋，对吗？但为了做到这点，你必须先学习许多技术。我们将看到，程序设计语言是简单的，其语法是严格定义的。但不幸的是，计算机是愚蠢的，极小的一个程序语法错误对于计算机来说也是致命的。而更糟糕的是，就算程序合乎文法，它也不一定如预期的那样完成计算任务。

程序设计需要耐心和专心，只有关注每个微小的细节才能避免沮丧的文法错误，只有严格的规划和

对规划的服从才能在设计中防止严重的逻辑错误。当你最终掌握程序设计的时候，你还将学到超越程序设计领域的许多有用知识。

让我们开始吧！



数、表达式和简单程序

第 2 章

在计算机诞生的初期，人们将其想象为处理数值的机器。实际上，计算机也善于进行数值处理。既然小学一年级老师最先讲的是数的运算，本书也从数开始。一旦了解计算机如何进行数值计算，只要将常识转换为程序语言符号，我们就可以设计出简单的计算机程序了。尽管如此，编写简单的程序也是需要原则的，因此本章的最后将介绍最基本的程序设计诀窍。

2.1 数和算术运算

数有多种形式，正数、负数、分数（也称有理数）和实数是最常见的数：

```
5    -5    2/3    17/3    #i1.4142135623731
```

其中第 1 个数是正数，第 2 个数是负数，接着是两个分数，最后一个是实数的非精确表示。

如同使用最简单的计算机，即计算器，在 Scheme 中你也可以对数进行加减乘除运算：

```
(+ 5 5)    (+ -5 5)    (+ 5 -5)    (- 5 5)    (* 3 4)    (/ 8 12)
```

其中前 3 个表达式要求 Scheme 执行加法运算，后 3 个表达式则分别是减法、乘法和除法运算。所有表达式都使用了括号，其中操作在前，接着是以空格隔开的操作数。

与算术、代数公式类似，Scheme 表达式也可以嵌套使用：

```
(* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))
```

Scheme 对这些表达式的计算与算术一样，首先将最内层的表达式计算为数值，然后是外面一层，以此类推：

```
  (* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))
= (* 4 (/ (* 8 3) 2))
= (* 4 (/ 24 2))
= (* 4 12)
= 48
```

由于每个表达式的形式皆为：

```
(operation A … B)
```

因此不存在哪部分先进行计算的问题。当 *A* … *B* 都是数的时候，就可以对表达式进行计算了，否则需要先对 *A* … *B* 进行计算，以

$$3+4 \cdot 5$$

为例，这是一个小学时就会遇到的表达式，在做了大量的练习后大家记得了在计算的时候应该先乘除后加减[1]。

[1] Scheme 语言的一个优点是我们知道在哪里放置或找到操作符，操作符总是紧挨着左括号，这在计算中非常重要，因为我们会需要许多操作符，不像算术和代数只使用了少数一些。

除了包括常见的简单数学运算符外，Scheme 还提供了一整套数学运算函数，以下是 5 个例子：

1. **(sqrt** *A*)计算 $\sqrt{A}$；
2. **(expt** *A B*)计算 A^B；
3. **(remainder** *A B*)计算整数 *A* 除以整数 *B* 的余数；
4. **(log** *A*)计算 *A* 的自然对数；
5. **(sin** *A*)计算弧度 *A* 的正弦值。

如果读者怀疑一个基本运算是否存在或欲了解其使用方式，请使用简单例子在 DrScheme 上进行测试。

关于数：Scheme 可以使用产生精确结果的基本操作对精确整数和有理数进行计算。因此，(/ 44 14) 的结果是 22/7。不幸的是，当涉及实数的时候，和其他程序语言一样，Scheme 在精度上做了折中考虑。例如，2 的平方根是一个实数而不是一个有理数，因此 Scheme 不得不使用非精确数来表示它：

```
  (sqrt 2)
=#i1.4142135623731
```

其中#i 警告程序设计者，计算结果是真正数值的一个近似表示。一旦一个非精确数成为计算的一部分，计算过程将以近似的方式进行，如：

```
  (- #i1.0 #i0.9)
= #i0.09999999999999998
```

而

```
  (- #i1000.0 #i999.9)
= #i0.10000000000002274
```

但从数学上看，两者的结果都应是 0.1，是相等的。因此一旦一个数是非精确的，系统应给出警告。

造成非精确的原因是用简化的方式表示 2 的平方根或如发现这样的数值。实际上这些数的十进制表示是无限长的（不含循环），而在一台计算机中，数的表示长度是有限的，因此只能表示这些数的一部分。如果将这些数表示为固定长度的有理数，其结果必然是非精确的。第 33 章将讨论非精确数是如何工作的。

为了集中精力学习与计算相关的重要概念而不是拘泥于这些细节，DrScheme 会尽量将数处理为精确数。如在 DrScheme 中输入 1.25，系统会将该数解释为一个精确的分数，而不是一个非精确的数值。当 DrScheme 的交互窗口显示一个如 1.25 或 22/7 这样的数值时，它就是一个对精确的有理数或分数进行计算的结果。仅当一个数的前缀为#i 时，它才是一个数的非精确表示。

习题

习题 2.1.1 查明 DrScheme 是否具备平方、计算一个角度的正弦值以及确定两个数的最大值的运算。

习题 2.1.2 在 DrScheme 中计算(sqrt 4)、(sqrt 2)和(sqrt -1)。再查明 DrScheme 是否包含计算一个角度的正切值的运算。

2.2 变量和程序

在代数中，可以用含变量的表达式阐明两个数之间的关系。变量是一个未知数的占位符(placeholder)。

例如，一个半径为 *r* 的圆盘的近似面积为[1]

$$3.14 \cdot \gamma^2$$

式中 *r* 是任意的正数。如果圆盘的半径为 5，则可以先将表达式中的 *r* 替换为 5，然后再对所得的表达式进行计算：

$$3.14 \cdot 5^2 = 3.14 \cdot 25 = 78.5。$$

一般来说，一个包含变量的表达式可以被认为是一条从给定值计算另一个数值的规则。

程序也是一种规则，它不仅告诉我们也告诉计算机应如何从一些数据产生另一些数据。一个大型的程序可能包含多个以某种方式组合起来的小程序，因此程序设计者在编写程序的时候如何给它们命名是非常重要的。对于上述程序，合适的名字是 *area-of-disk*。使用该名字，可以将计算圆盘面积的程序表示如下：

```
(define (area-of-disk r)
  (* 3.14 (* r r)))
```

上两行程序指明 *area-of-disk* 是一条规则，*r* 是其惟一的输入，一旦知道了 *r* 的值，程序的结果或输出就是(* 3.14 (* *r r*))。

程序将一些基本操作组合在一起。在上例中，*area-of-disk* 仅仅使用了一个基本操作，即乘法。实际上，定义程序可以使用任意数目的操作。函数一旦被定义，此后就可以如同基本函数那样被使用。对函数名右边列出的变量，我们必须提供一个输入，也就是说，我们可以编写表达式，其中操作是 *area-of-disk*，其后跟着一个数值：

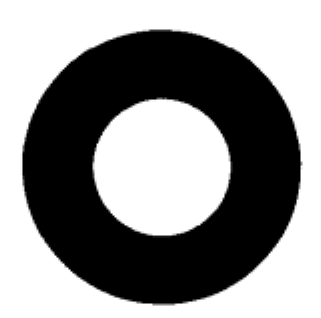
```
(area-of-disk 5)
```

其意为将 *area-of-disk* 应用于数值 **5**。

应用一个已定义的函数（如 *area-of-disk*）的过程是先拷贝名为 *area-of-disk* 的表达式并将其中的变量(*r*)替换为相应的数值(**5**)，然后再进行计算：

```
  (area-of-disk 5)
= (* 3.14 (* 5 5))
= (* 3.14 25)
= 78.5
```

很多程序的输入多于一个，计算圆环（中心有一个洞的圆盘）面积的程序就是一个例子：



我们知道圆环的面积是外盘的面积减去内盘的面积，这意味该程序需要两个未知量：外盘的半径 *outer* 和内盘的半径 *inner*，因此，计算圆环的面积的程序可以写成：

```
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```

[1] 通常我们说一个圆的面积，但从数学上说，圆仅是一个圆盘的外边缘。

此 3 行代码表示 *area-of-ring* 是一个程序，该程序有两个输入，*outer* 和 *inner*，并且程序结果是(*area-of-disk outer*)和(*area-of-disk inner*)之差。换句话说，*area-of-ring* 使用了 Scheme 的操作和前面已经定义过的函数。

area-of-ring 有两个输入，如

```
(area-of-ring 5 3)
```

该表达式的计算和(*area-of-disk* 5)的计算是一样的，先从函数定义拷贝表达式，然后将其中的变量替换为输入的数值：

```
  (area-of-ring 5 3)

= (-(area-of-disk 5)
    (area-of-disk 3))

= (-(* 3.14 (* 5 5))
    (* 3.14 (* 3 3)))
= ...
```

接下去是一般的算术运算。

习题

习题 2.2.1 定义程序 *Fahrenheit->Celsius*[1]，输入为华氏温度值，输出为等值的摄氏温度值。请查阅化学或物理书籍了解温度的转换公式。

设计出函数后，使用教学软件包 convert.ss 对所设计的函数进行测试。该教学软件包提供了 3 个函数 *convert-gui*、*convert-repl* 和 *convert-file*。第一个函数用于创建图形用户界面，请按以下方式调用

```
(convert-gui Fahrenheit->Celsius)
```

该表达式将创建一个包含按钮和滚动条的新窗口。

第二个函数仿真一个交互式的窗口，它要求用户输入一个华氏温度值，该数值由程序读入后，对其计算并打印，调用方式为：

```
(convert-repl Fahrenheit->Celsius)
```

最后一个函数处理的是数据文件，使用该函数之前，需要先创建一个数值文件，文件中的数值由空格或换行符分隔。函数读入文件后，对数值进行转换，并将结果写到另一个新文件中，调用方式如下：

```
(convert-file "in.dat" Fahrenheit->Celsius "out.dat")
```

这里假定所创建的数值文件的名字为 in.dat，写入结果的文件的名字为 out.dat。要了解更多的信息，请使用 DrScheme 的 Help Desk 来查找关于教学软件包 convert.ss 的信息。

习题 2.2.2 定义程序 *dollar->euro*，该程序输入为美元值，输出为等价的欧元值。请查阅报纸了解美元对欧元的汇率。

习题 2.2.3 定义程序 *triangle*，该程序输入为一个三角形的底和高的长度，输出为三角形的面积。请查阅平面几何书籍了解三角形面积的计算公式。

习题 2.2.4 定义程序 *convert3*，输入为 3 个数，分别代表一个数值的个位、十位、百位上的数，程序输出为相应的数值。例如，

```
(convert3 1 2 3)
```

的输出为 321。请查阅代数书籍了解该转换过程。

---

[1] 箭头的输入方法是先输入——后输入>。

习题 2.2.5　经典代数书籍往往要求读者分别在 n=2、n=5 和 n=9 时计算如下公式：

$$\frac{n}{3}+2$$

使用 Scheme 可以先将该上述表达式表示为函数，然后再应用于不同的参数，下面就是相应的程序：

```
(define (f n)
  (+ (/ n 3) 2))
```

请先手工计算 n=2、n=5 和 n=9 时表达式的值，然后使用 DrScheme 的按步执行功能（stepper）计算表达式的值并对照各自所得的结果。

请将下述公式转换为程序，并手工计算 $n = 2$ 和 $n = 9$ 时表达式的值，再在 DrScheme 下运行该程序，并对结果进行比较。

1. $n^2 + 10$
2. $(1/2) \cdot n^2 + 20$
3. $2-(1/n)$

2.3　字处理问题

程序设计者一般较少处理诸如将数学公式转换为程序这样的问题，他们通常要处理的问题往往缺乏形式化的描述、包含不相关甚至含糊的信息。程序设计者的第一个任务就是从问题中提取相关信息然后用合适的表达式阐明。以下是一个典型例子：

XYZ 公司所有雇员的报酬都是每小时 12 美元。通常每个雇员每周工作 20 到 65 小时。试编写一个程序按照雇员的每周工作时数计算其周薪。

最后一句话提出了实际的任务：编写一个程序根据某些数值计算另一个数值。具体地说，程序的输入是一个数值，即每周工作时数，输出是另一个数值，即周薪。第一句话说明计算是如何进行的，但没有对其明确阐明，在此，这不会引起问题。容易看出，如果一个雇员工作了 h 小时，他的周工资就是

$$12 \cdot h。$$

知道了规则，用 Scheme 语句写出来就是：

```
(define (wage h)
  (* 12 h))
```

该程序的名字为 *wage*，参数 h 是一个雇员的每周工作时数，结果为相应的周薪，即(* 12 *h*)。

习题

习题 2.3.1　在乌托邦计算所得税的税率是固定的，为毛收入的 15%。试编写程序 *tax*，按照雇员的毛收入计算所得税。并编写程序 *netpay*，计算雇员的税后所得。假定雇员的每小时工资为 12 美元。

习题 2.3.2　假定当地超级市场需要一个程序计算一袋硬币的价值。编写程序 *sum-coins*，其输入为钱袋中 1 美分、5 美分、10 美分和 25 美分的硬币数，输出为钱袋中硬币的价值总额。

习题 2.3.3　某旧式电影院有一个简单的利润计算：每张电影票价格为 5 美元，每场电影放映的成本为 20 美元，再加上每位观众的耗费 0.5 美元。试编写程序 *total-profit*，输入为一场电影的观众数，输出为电影院的净收入。

2.4 错 误

编写 Scheme 程序必须遵循一些规则[1]。这些规则在计算机能力和人们的行为之间进行折衷。幸运的是，Scheme 定义和表达式的结构是直观的，表达式或者是原子表达式，即数和变量；或者是复合表达式，它以“（”开始，接着是操作，然后是其他的表达式，最后以“）”结束。复合表达式中的每一个公式前面都至少有一个空格，换行符也可以，有时使用换行符还可以增加程序的可读性。

函数定义的方式为：

```
(define (f x … y)
  an-expression)
```

可见函数定义是字符和表达式的序列，即左括号“（”、define、由空格分隔的非空名字序列、一个表达式和右括号“）”。其中 *f*、 *x*、 …、 *y* 分别是函数名和参数。

语法错误[2]：并不是所有带括号的表达式都是合法的 Scheme 表达式。例如，（10）是一个带括号的表达式，但不是合法的 Scheme 表达式，因为 Scheme 不认为一个数应被包含在括号中。类似地，(10+20) 也是不合法的，因为 Scheme 规则要求操作先于操作数出现。最后，下面两个定义也是不合法的：

```
(define (P x)
  (+ (x) 10))
(define (Q x)
  x 10)
```

其中第一个函数包含了一对括号，它们将 *x* 包含其中，而 *x* 是变量，不是复合表达式。第二个函数则包含有两个原子表达式，*x* 和 10，不是一个。

按下 DrScheme 的 Execute 按钮后，Scheme 程序设计环境首先会按照 Scheme 的语法规则检查程序定义是否合法。如果 Definitions 窗口中程序的某一个部分不合法，DrScheme 将提示相应的语法错误，给出相应的错误消息并高亮显示出现错误的地方，否则将允许函数的使用者在交互窗口中对表达式进行计算。

习题

习题 2.4.1 在 DrScheme 中逐个计算如下表达式：

```
(+ (10) 20)
(10 + 20)
(+ +)
```

阅读并理解错误消息。

习题 2.4.2 在 DrScheme 的 Definitions 窗口中逐个输入下述语句并点击 Execute 按钮：

```
(define (f 1)
  (+ x 10))
```

---

[1] 对于其他程序设计语言来说，如电子表格、C、字处理软件中的宏，这一点同样成立。Scheme 比这些语言中的大多数都简单并且易于被计算机了解。不幸的是，对那些习惯中缀表示法，如 5+4 等人来说，Scheme 的前缀表示法是复杂的。但稍加练习便可克服这种不习惯。

[2] 在第 8 章中，我们会了解到为什么这种错误被称作语法错误。

```
(define (g x)
  + x 10)
(define h(x)
  (+ x 10))
```

阅读错误消息，进行适当修改，直到所有的定义都合法为止。

运行错误：Scheme 表达式的计算过程与代数、算术上的计算法则一样。当遇到新的操作时，需要对计算法则进行扩展，这里先直观说明，第 8 章再进行严格讨论。重要的是，并不是所有合法的 Scheme 表达式都有结果。一个简单的例子是(/ 1 0)。类似地，如果函数+定义如下：

```
(define (f n)
  (+ (/ n 3) 2))
```

我们就不能让 DrScheme 去计算(*f* 5 8)。

当让一个合法的 Scheme 表达式进行除数为零的除法运算，或进行其他无意义的算术运算，或当一个程序的输入数目有错时，DrScheme 将停止计算过程并给出运行错误消息。通常是在交互窗口中显示一个解释并高亮显示发生错误的表达式。高亮显示的表达式是引发错误的原因。

习题

习题 2.4.3　在 DrScheme 的交互窗口中计算下面文法上合法的 Scheme 表达式并阅读错误消息：

```
(+ 5 (/ 1 0))
(sin 10 20)
(somef 10)
```

习题 2.4.4　在 DrScheme 的 Definitions 窗口中输入下面文法上正确的 Scheme 表达式并点击 Execute 按钮：

```
(define (somef x)
  (sin x x))
```

接着在交互窗口中，计算下列表达式并阅读错误消息：

```
(somef 10 20)
(somef 10)
```

观察 DrScheme 中高亮显示的表达式。

逻辑错误：一个良好的程序设计环境可以帮助程序设计者发现语法和运行错误。本节的练习说明 DrScheme 如何捕捉语法和运行错误。但程序设计者也可能制造逻辑错误。一个逻辑错误可能不会激发任何错误消息，但计算所得的结果却是错误的。考虑上一节提到的程序 *wage*，如果程序设计者将其定义为

```
(define (wage h)
  (+ 12 h))
```

程序的每次运行也会产生一个数值。如果计算(*wage* 12/11)，甚至可能得到正确的结果。只有细心和系统地设计程序，程序设计者才能捕获到此类错误。

2.5 设计程序

上面章节说明程序设计需要考虑许多步骤，需要确定问题描述中哪些信息是相关的、哪些是可以忽略的，需要了解程序的输入和输出以及它们之间的关系。我们必须知道或查明 Scheme 是否提供所需的处理数据的基本操作，如果没有，还必须设计一些辅助函数来实现它们。最后，一旦编写了程序，还必须验证、测试它是否能完成预期的任务。该过程可能会暴露一些语法错误、运行问题甚至是逻辑错误。

要解决这些表面上混乱的情况，必须建立并遵循一套设计诀窍，即规定完成任务的顺序以及每步如何进行[1]。基于到目前为止所得到的经验，设计一个程序至少需要如下 4 个步骤：

理解程序的目的：程序设计的目标是创建一个接收输入并产生结果的机制。因此在开发程序时应该给每一个程序一个有意义的名字，并且说明输入数据和所产生的数据的类型，这称为程序的合约。下面是程序 *area-of-ring* 的合约：

```
;; area-of-ring : number number -> number
```

其中分号表示该行是一个注释。合约包含两个部分，冒号的左边是程序的名字，右边是输入和输出的类型，输入和输出之间用箭头隔开[2]。

一旦有了合约，就可以在程序中加入函数头部，函数头部复述了程序的名字，同时给每个输入一个不同的名字。这些名字是（代数）变量，是程序的参数[3]。

下面是程序 *area-of-ring* 的合约和函数头部：

```
;; area-of-ring : number number -> number
   (define (area-of-ring outer inner) ...)
```

它们表示程序的第一个输入为 *outer*，第二个输入为 *inner*。

最后，基于合约和参数，简要阐明一下程序的用途说明，它是程序要完成的任务的简短注释。对于大多数程序，一到两行就足够了，更大的程序则需要更多的信息来说明其用途。

现在完整的程序开头如下：

```
;; area-of-ring : number number -> number
;; 计算一个半径为 outer，洞的半径为 inner 的圆环的面积
(define (area-of-ring outer inner) ...)
```

提示：如果问题表述包含了数学公式，公式中不同变量的数目可能就是程序的输入数。

为了将给定的事实与要计算的数据分开，我们必须仔细检查问题表述。如果给定的是一个固定数值，它可能要在程序中出现。如果给定的是一个稍后需要确定的未知数，它就是一个输入，而问题表述中的询问（或要求）则提示了程序的名字。

例子：为了更好地了解程序要计算什么，需要构造一些输入并确定输出到底是什么。例如，对于输入 5 和 3，程序 *area-of-ring* 的计算结果应为 50.24，这是因为程序的输出是外圆盘的面积与内圆盘面积之差。

在用途说明中加入例子：

```
;;  area-of-ring : number number -> number
;; 计算一个半径为 outer，洞半径为 inner 的圆环的面积
```

[1] 正如我们将看到的那样，这个顺序并不是完全固定的。这可以在某些情况下改变这些顺序。

[2] 输入箭头的方法是先键入-再键入 >。

[3] 一些人称其为形式参数或输入变量。

```
;; 例子: (area-of-ring 5 3) 的结果为 50.24
(define (area-of-ring outer inner) ...)
```

在编写程序体之前构造例子从多方面看来都是有益的。首先，它是惟一可靠的在程序测试中发现逻辑错误的途径。如果借助最终得到的程序来构造例子，有可能会轻信程序，因为运行程序比预测它会做什么容易得多。第二，例子使我们思考数据计算过程，这对于将遇到的复杂程序体的设计是至关重要的。最后，例子是用途说明的非正式表达。此后的程序读者，如教师、同事还有程序购买者会喜欢这些抽象概念的具体说明。

程序体：最后必须阐明程序体，即必须将函数头部中的“...”替换为表达式。该表达式使用 Scheme 中的基本操作和已定义或即将定义的程序，由参数计算出结果。

只有理解了如何从给定的输入计算出结果，才可能阐明程序体。如果输入和输出的关系由数学公式给出，只要将数学公式转换为 Scheme 表达式即可。如果给定的是一个书面叙述的问题，我们必须细心地挖掘其中的信息并构造相应的表达式。最后，观察并理解如何从特定的输入得到输出的例子可能对程序体的设计也会有所帮助。

在我们所讨论的例子中，计算任务是一个非正式说明的公式，它使用了先前定义的程序 *area-of-disk*，下面是它的 Scheme 翻译：

```
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```

测试：在完成了程序定义之后，还必须测试程序。至少应该确定对于给定的例子，程序计算所得的结果与预期数值是否相符。为了简化程序测试过程，通常可以在 Definitions 窗口的下面如同添加等式一样添加一些例子。然后，按下 Execute 按钮，计算它们，并观察对于这些例子程序是否正常工作。

测试不能保证程序对所有可能的输入都产生正确的输出，因为可能的输入数目通常是无限的。但测试可以揭示语法错误、运行问题以及逻辑错误。

对于错误的程序输出，必须特别关注程序例子。有可能例子本身就是错误的，也有可能程序包含了逻辑错误，也有可能例子和程序都有错误。不管是何种情况，都必须再次历经程序开发的每一步。

图 2.1 说明了按照上述诀窍开发程序所得到的结果。图 2.2 以表格形式总结了设计诀窍，我们设计程序时都得参考这些诀窍。

```
;; 合约: area-of-ring : number number -> number
;; 用途: 计算一个半径为 outer，其中洞的半径为 inner 的圆环的面积
;; 例子: (area-of-ring 5 3) 的计算结果为 50.24
;; 定义: [函数头部的精化]
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))

;; 测试:
(area-of-ring 5 3)
;; 预期的值
50.24
```

图 2.1　设计诀窍：一个完整的例子

设计诀窍并不是魔法，它并不能解决程序设计过程中所遇到的所有问题，它提供的是完成程序设计过程中必经步骤的指导。在程序设计中最富有创新性和最困难的一步是程序体的设计。它依赖于我们阅读和理解书面材料的能力，依赖于我们获取数学关系的能力，依赖于我们所掌握的基本事实。上述任何

一点对计算机程序的开发者来说都不是特殊的，而所使用的知识对不同的应用领域来说却是有差异的。本书的其余部分将说明如何完成这最困难的一步。

领域知识：阐明程序体通常需要与问题相关的知识，这种形式的知识称为领域知识（domain knowledge）。它可能来自简单或复杂的数学，如算术和微分方程，或来自非数学学科，如音乐、生物学、土木工程和艺术等。

一个程序设计者不可能了解所有计算应用领域，但他必须准备去了解不同应用领域的语言，以便和领域专家沟通，所使用的语言可能是数学。但在有些情况下，程序设计者必须发明一种语言，特别是描述应用领域数据的语言。因此，程序设计者必须对计算机语言的所有可能性有充分的理解。

阶段	目标	任务
合约、 用途说明和 函数头部	给函数命名； 指定输入和输出的类型； 描述函数的用途说明； 阐明函数头部	给函数起一个合适的名字 • 以函数需要的未知数为线索研究问题； • 给每个输入起一个名字，如果可能的话，使用在问题描述中给定的名字； • 使用选择的变量名描述函数应该产生什么结果； • 阐明合约和函数头部： ;; *name : number ...->number* ;; x1 开始计算... （**define** (*name x1*...)...）
例子	通过例子刻划输入和输出之间的关系	检查问题表述得到例子 • 计算例子； • 如果可能的话，检查计算结果； • 构造例子
主体	定义函数	阐明函数是如何计算它的结果的 • 使用 Scheme 基本运算、其他函数和变量构造 Scheme 表达式； • 如果可以的话，翻译问题描述中的数学公式
测试	发现错误（拼写错误和逻辑错误）	将函数应用于例子中的输入数据 • 检查结果与预期值是否相符

图 2.2 设计决窍一览



第 3 章

程序就是函数加上变量定义

通常，一个程序不仅包含一个，而是包含多个定义。例如，上一章的 *area-of-ring* 程序就包含了两个定义，一个为 *area-of-ring*，另一个为 *area-of-disk*，两者都被称为函数定义。使用数学术语，可以说一个程序包含若干个函数。其中第一个函数，即 *area-of-ring*，是我们实际上想使用的，因此被称为主函数，而第二个函数，*area-of-disk*，被称为辅助函数。

使用辅助函数不仅使程序设计过程易于管理并且使最后得到的程序易于阅读。试比较一下 *area-of-ring* 程序的两个不同版本

```
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```

```
(define (area-of-ring outer inner)
  (- (* 3.14 (* outer outer))
     (* 3.14 (* inner inner))))
```

左边的定义包含了辅助函数。它将一个原先较大的问题分解为若干较小而容易解决的子问题，定义提示了区域的面积等于整个盘的面积减去洞的面积。与之相反，右边的定义则需要读者通过计算推导出两个子表达式的功能。另外，右边不得不以一个单一的程序块给出函数的定义，因此无法从求解过程分解中受益。

设计一个像 *area-of-ring* 这样小型的程序，两种程序风格的差异是很小的。而对于大型程序，使用辅助函数不仅是一种选择而且是必需的。就算编写一个简单的程序，也应该考虑将其分解为若干较小的子程序，然后在需要的时候再组合在一起。虽然现在还没有开始大型程序的开发，本书仍然将通过同时展示两种不同版本的程序使读者对这一思想有所认识。

本章内容安排如下，第 1 小节使用一个商业界的例子对两种程序开发风格进行比较，说明将一个程序分解为若干函数定义可以使我们更有信心确保整个程序的正确性；第 2 小节引入变量定义概念，它是程序开发过程中另一个重要的因素；最后一小节是一些练习。

3.1 函数复合

考虑下面的问题：

假定一个影院的拥有者（业主）可以自由制定电影票的价格。显然票价越高，看电影的人就越少。在最近的一次试验中，他测定了票价和观众数之间的关系。当票价为 5 美元时，有 120 人观看了电影，当影票的价格调低了 0.1 美元后，观众增加了 15 位。不幸的是，观众的增加也增加了电影院的成本。每放映一场电影需要支付 180 美元给供片商，而每位观众还要有 4 美分的开销。现在，想知道电影放映的利润和票价之间的确切关系，并由此确定一个利润最高的票价。

尽管问题非常清楚，但如何解决它我们尚不清楚。目前所能说的就是几个因素间的相互依赖关系。

当遇到这种情况时，最好是先分析一下依赖关系：

1. 利润是收入和成本之差。
2. 收入由票房惟一确定，它是票价乘以观众数。
3. 成本由固定成本（180 美元）与依赖于观众数的变动成本两部分组成。
4. 观众数和票价之间的关系。

接着对上述依赖关系给出函数表示。

下面是以合约、函数头部和用途说明开始的函数 *profit* 的描述：

```
;; profit : number -> number
;; 对于给定 ticket-price1，利润是收入和成本之差
(define (profit ticket-price) ...)
```

利润之所以依赖于票价是因为收入和成本都依赖于票价。下面是其他三个函数的说明：

```
;; revenue : number -> number
;;对于给定 ricket-price，计算收入
(define (revenue ticket-price) ...)
;; cost : number -> number
;; 对于给定 ticket-price，计算支出
(define (cost ticket-price) ...)
;; attendees : number -> number
;;对于给定 ticket-price，计算观众数
(define (attendees ticket-price) ...)
```

其中每个函数的用途说明都是问题表述中某一部分的粗略转译。

习题

习题 3.1.1 为每个函数构造计算实例。例如，确定当票价为 3 美元、4 美元或 5 美元时有多少人愿意买票看电影。使用实例可以了解从票价计算观众数的一般规则。在需要的时候还可以尝试更多的例子。

习题 3.1.2 使用习题 3.1.1 的结果计算当票价为 3 美元、4 美元和 5 美元时放映电影的成本，并进一步计算在上述票价下放映电影的收入，最后计算业主在每种情况下的利润。思考题：若要使利润最大，要将票价定为多少？

写下函数的一些基本材料并计算了若干实例后，接着就可以将函数中的"..."替换为 Scheme 表达式。图 3.1 左边一栏包含了上述 4 个函数的完整定义。正如问题分析和用途说明所提示的那样，*profit* 函数的值是 *revenue* 函数的值和 *cost* 函数的值之差。而 revenue 和 cost 函数的计算都依赖于票价。计算收入时，程序先算出给定票价下的观众数，然后将其乘以票价。类似地，计算支出时，程序将固定开销加上可变开销，其中可变开销是观众数乘以 4 美分。最后观众数的计算也遵循问题描述，即票价为 5 美元时，观众的数目是 120，票价每减少 10 美分，观众的数目就增加 15。

如果不想对问题表述中的每种依赖关系都设计一个函数，也可以尝试将票价和业主的收益用一个简单的公式加以表示。很容易验证图 3.1 右栏中的程序对于给定的票价将产生与图 3.1 左栏中的程序相同的结果。尽管如此，它们还是有差别的，如左栏中的程序的编排体现了程序的含义，而右栏中的程序的含义却几乎不可能被读者所理解。更糟糕的是，如果要求对程序的某一部分进行修改，如对票价和观众数之间的关系重新定义，修改左栏中的程序可以在较短的时间内完成，而右栏中的程序的修改就需要较多的时间了。

```
;; 如何设计程序
(define (profit ticket-price)
  (- (revenue ticket-price)
     (cost ticket-price)))
(define (revenue ticket-price)
(*  (attendanees ticket-price) ticket-price))
 (define (cost ticket-price)
  (+ 180
     (* .04 (attendanees ticket-price))))
 (define (attendanees ticket-price)
  (+ 120
     (* (/ 15 .10) (- 5.00 ticket-price))))
```

```
;; 不要这样设计程序
(define (profit price)
  (- (* (+ 120
           (* (/ 15 .10)
              (- 5.00 price)))
      price)
    (+ 180
   (* .04
      (+ 120
         (* (/ 15 .10)
            (- 5.00price))))))
```

图 3.1 程序 *profit* 的两种表述方式

因此，基于上述经验，总结出第一条也是最重要的程序设计原则如下：

> **辅助函数原则**
>
> 对在表述中所提到的或在进行实例计算中所发现的每种依赖关系都使用一个辅助函数进行明确表达。

在进行程序设计时，有时会发现许多所需要的函数已经出现在求解其他问题的程序中，事实上，我们已经遇到过这样的例子，如 *area-of-disk*。我们一般先列出一个函数表并分别进行设计。以后可能会发现其中的一些函数，如上例中的 *attendees*，在许多定义中都是有用的，因此函数之间的关系是网状的。

习题

习题 3.1.3 分别使用在图 3.1 两栏中定义的函数计算当业主将票价定为 3 美元、4 美元和 5 美元时的利润，确保其结果和习题 3.1.2 中所预期的相同。

习题 3.1.4 研究了放映电影的开销结构后，业主发现了几种降低开销的方法。其中之一是，取消固定成本，对每个观众支付 1.5 美元给供片商，请修改程序以反映这种变动。修改程序后，用 3 美元、4 美元和 5 美元票价测试程序并对结果进行比较。

3.2 变量定义

如果一个数值在程序中多次出现，应该使用变量定义给它一个名字。变量定义将一个名字与一个值相关联。例子之一是 3.14，通常用π来代表，相关的变量定义语句为：

```
(define PI 3.14)
```

现在，每次引用 *PI*，DrScheme 都会将它替换为 3.14。

使用名字表示一个常量的好处是可以方便地将一个数值替换为另一个不同的数值。假定已有一个包含 *PI* 的定义的程序，现在想使用一个更精确的π的近似值，则可将定义改为

```
(define PI 3.14159)
```

则程序中任何对 *PI* 的引用都会得到替换。如果没有一个如同 *PI* 这样的表示π的名字，则我们不得不在程序中寻找 3.14 的所有出现处并将其改为 3.14159。

将上述观察表示为第 2 条程序设计原则：

变量定义原则

给频繁使用的常量定义一个名字并在程序中使用。

最初，由于程序规模较小，可以不对多数的常量进行变量定义。但当编写规模较大的程序时，应该尽可能使用变量定义。正如将看到的那样，改动具有单一控制点的能力对于变量和函数定义来说是非常重要的。

习题

习题 3.2.1 给出图 3.1 中所有常量的变量定义并用它们的名字替换在程序中出现的常量。

3.3 函数复合练习

习题 3.3.1 由于美国使用的是英制单位，而世界上其他国家一般使用的是公制单位。因此，在世界各地旅游的人士以及和外商进行商业往来的公司常常需要在这两种度量衡之间进行转换。下面是 6 种主要英制长度单位和公制单位的换算表[1]：

英制				公制		
1 inch				=	2.54	cm
1 foot	=	12	in.			
1 yard	=	3	ft.			
1 rod	=	$5(\frac{1}{2})$	yd.			
1 furlong	=	40	rd.			
1 mile	=	8	fl.			

请设计函数 *inches->cm*、*feet->inches*、*yards->feet*、*rods->yards*、*furlongs->rods* 和 *miles->furlongs*。请进一步设计函数 *feet->cm*、*yards->cm*、*rods->inches* 和 *miles->feet*。

提示：尽可能地重用函数并使用变量定义对常量进行说明。

习题 3.3.2 设计程序 *volume-cylinder*。给定圆柱体半径和高度，该程序计算圆柱体的体积。

习题 3.3.3 设计程序 *area-cylinder*。给定圆柱体半径和高度，该程序计算圆柱体的表面积。

习题 3.3.4 设计程序 *area-pipe*。计算一个管道(管道是一中空的圆柱体)的表面积。程序的输入为管道的内半径、长度和厚度。请设计两种版本，一种版本的程序仅包含单一函数，另一种版本的程序包含若干个函数定义。考虑一下哪一种版本的程序更有用。

习题 3.3.5 设计程序 *height* 以计算一枚火箭升空后在给定时刻所到达的高度。假定火箭的加速度 g 为常量，在 t 时刻速度为 g t，高度为 1/2 * v * t，其中 v 是火箭在 t 时刻的速度。

习题 3.3.6 回忆一下习题 2.2.1 中的程序 *Fahrenheit->Celsius*，该程序的输入为华氏温度，输出为等价的摄氏温度。设计程序 *Celsius->Fahrenheit*，其输入为摄氏温度，输出为等价的华氏温度。现考虑

[1] 请参见 Weights 和 Measurements 1993 年撰写的 *The World Book Encyc lopedia* 一书

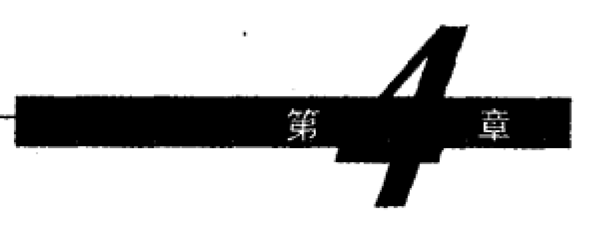
函数

```
;; I : number -> number
;; 将华氏温度转换为摄氏温度再转换回华氏温度
(define (I f)
  (Celsius->Fahrenheit (Fahrenheit->Celsius f)))
```

请分别手工和使用 DrScheme 的按步执行方式计算(*I* 32)。并思考从这两个函数的复合中你得到的启发是什么？



条件表达式和函数



对于许多问题，计算机程序必须以不同的方式处理不同的可能情况。如，一个游戏程序必须确定一个物体的移动速度是否在某个范围之内或确定一个物体是否位于屏幕某个特定的区域。而对于一个控制电机运行的程序，则可能需要描述阀门什么情况下应被打开。为了处理这些情况，我们必须使用一种方式来阐述所述条件为真或为假，即需要一种新的数据类型。通常，该类型的值被称为布尔值（或真值）。本节介绍布尔类型、计算为布尔值的表达式以及依赖于布尔值进行计算的表达式。

4.1 布尔类型和关系

考虑以下问题：

XYZ 公司给雇员的报酬是每小时 12 美元。通常一个员工每周工作 20 到 65 小时。如果一个雇员的每周工作时数在上述范围内，试编写程序确定其周工资。

斜体突出显示了相对 2.3 节中的问题所新增的部分，它表示了程序必须以某种方式处理它的输入，如果输入在给定的范围之内，按常规进行，否则，则需考虑其他的计算公式。简而言之，就像人们对不同的情况分别进行推理一样，程序使用条件表达式对不同的情况进行计算。

条件并不是一种新的概念，数学所说的断言的真假就是一种条件。例如，一个数可能等于、小于或大于另一个数。因此如果 x 和 y 是数，则关于 x 和 y 的如下：

1. $x = y$： x 等于 y；
2. $x < y$： x 严格小于 y；
3. $x > y$： x 严格大于 y。

对于任何一对给定的（实）数，上述断言有且只有一个是正确的。如果 $x = 4$ 且 $y = 5$，则只有第 2 个断言为真，其他均为假。一般地，一个断言对于变量的某些取值为真，其余取值为假。

除了确定一个基本断言在给定的情况下是否为真以外，有时确定断言的组合是否为真也很重要。考虑以上 3 个断言，它们可以以下面的方式组合在一起：

1. $x = y$ 且 $x < y$ 且 $x > y$;
2. $x = y$ 或 $x < y$ 或 $x > y$;
3. $x = y$ 或 $x < y$。

第 1 个复合断言是假的，因为不管 x 和 y 的值为何，3 个断言中有 2 个必定为假，因此其组合也为假；另外，不管 x 和 y 取何值，第 2 个复合断言为真；最后，第 3 个复合断言，在一些情况下为真，在另一些情况下为假，如当 $x = 4$、$y = 4$ 或 $x = 4$、$y = 5$ 时为真，在 $x = 5$ 、$y = 3$ 时为假。

与数学语言一样，Scheme 有自己表示真假的词汇、有陈述基本断言的词汇、有将基本断言组合为复合断言的词汇。在 Scheme 中，真表示为 true，假表示为 false。如果断言涉及两个数的关系，通常会使用关系操作，如=、<和>等。

上述三个数学断言的转换遵循大家所熟悉的先写左括号，再写操作符，然后是参数，最后是右括号这样的 Scheme 表达式结构：

(= x y)：x 等于 y；
(< x y)：x 严格小于 y；
(> x y)：x 严格大于 y。

以后大家还会遇到诸如<=和>=这样的关系操作符。

比较两个数值大小的 Scheme 表达式和其他 Scheme 表达式一样有一个结果，但其结果不是数值，而是 true 或 false。当关于两个数的 Scheme 断言为真时，值为 true，如

```
  (< 4 5)
= true
```

断言为假时，值为 false，如

```
  (= 4 5)
= false
```

复合条件在 Scheme 中的表示也很自然。如果要把(= x y)和(< y z)组合在一起，表示当两个条件为真时复合断言为真，可以写成：

```
(and (= x y) (< y z))
```

类似地，如果想表示至少两个条件之一为真时，复合断言为真，可以写成：

```
(or (= x y) (< y z))
```

最后，下述表达式

```
(not (= x y))
```

表示断言的否定为真[1]。

与基本条件一样，复合条件的计算结果也是 true 或 false。考虑下列复合条件：

```
(and (= 5 5) (< 5 6))
```

它包含了两个基本断言，(= 5 5)和(< 5 6)，两者的计算结果都为 true，因此 and 表达式的计算过程如下：

```
  ...
= (and true true)
= true
```

最后一步计算的结果为 true 的原因是：如果 and 表达式两个部分的值皆为 true，则整个表达式的值为 true；反之，如果两个断言之一为 false，则 and 表达式的值就是 false，如：

```
  (and (= 5 5) (< 5 5))
= (and true false)
= false
```

or 和 not 的计算规则与 and 相类似。

下面一些章节将解释为何程序设计需要明确地对条件进行陈述和推理。

习题

习题 4.1.1　下述 Scheme 条件的结果是什么？

```
(and (> 4 3) (<= 10 100))
```

---

[1] 实际上，and、or 和 not 是不同的操作，但我们现在暂时忽略它们细微的不同。

```
(or (> 4 3) (= 10 100))
(not (= 2 3))
```

习题 4.1.2 下面表达式的结果是什么？

```
(> x 3)
(and (> 4 x) (> x 3))
(= (* x x) x)
```

请分别考虑(a) x =4、(b) x =2 以及(c) x =7/2 时的值。

4.2 函数和条件测试

下面是一个简单的对变量取值进行测试的函数：

```
;; is-5? : number ->boolean
;; 确定 n 是否等于 5
(define (is-5? n)
  (= n 5))
```

该函数仅当输入为 5 时，值为 *true*。在函数的合约中包含了一个新的要素 *boolean*。与 *number* 一样，*boolean* 是 Scheme 内建的一种数据类型。不同的是，*boolean* 仅包含两个值，true 和 false。

下面是一个稍微有点意思的输出类型为 *boolean* 的函数：

```
;; is-between-5-6? : number -> boolean
;; 确定 n 的值是否位于 5 和 6 之间（不包括 5 和 6）
(define (is-between-5-6? n)
  (and (< 5 n) (< n 6)))
```

如果输入的值介于 5 和 6 之间（不包括 5 和 6），函数输出结果为 true。理解此类函数的一种较好的方式是认为函数划分了数轴上的一个区间：



区间边界：一个以“（”或“）”标识的区间是不包含边界的，而以“[”或“]”标识的区间则包含边界。

下述函数刻划了一个较为复杂的区间：

```
;; is-between-5-6-or-over-10? : number -> boolean
;; 确定 n 是否介于 5 和 6 之间（不包括 5 和 6）或者大于等于 10
(define (is-between-5-6-or-over-10? n)
  (or (is-between-5-6? n) (>= n 10)))
```

对于数轴上两个区间内的任何数值，函数返回 true：



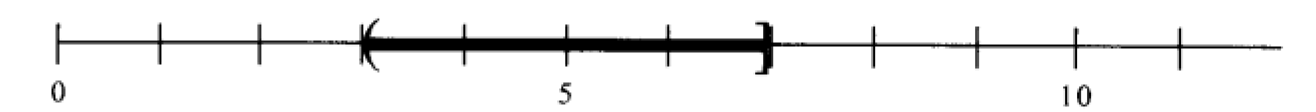
左边区间是 5 和 6 之间，但不包括 5 和 6 的任何数值，右边是从 10 开始且包括 10 的无限区间，数轴上两个区间的点都满足函数 *is-between-5-6-or-over-10* 中的条件表达式。

上述 3 个函数对数值进行了条件测试。为了设计或理解此类函数，必须理解区间和它们的组合（也称为区间的并）。以下是练习相关技巧的习题。

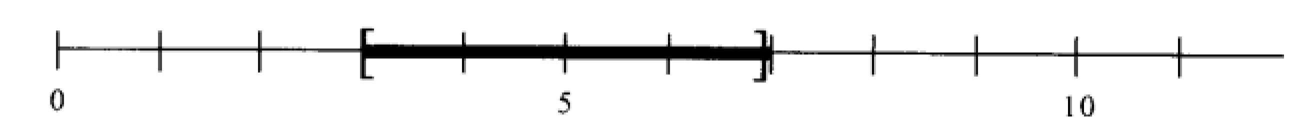
习题

习题 4.2.1　将下面数轴上的 5 个区间转换为 Scheme 函数，这些函数接受一个数值，当数值位于区间内时返回 true，否则返回 false。

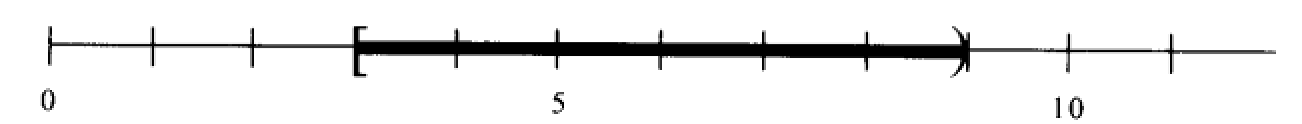
1. 区间(3，7)：



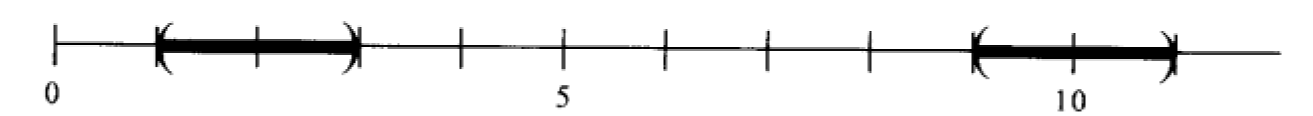
2. 区间[3，7]：



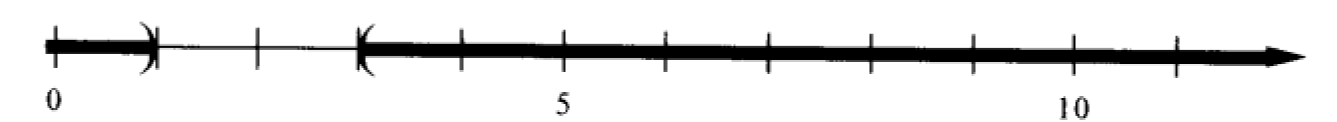
3. 区间[3，9]：



4. 区间(1，3)和(9，11)的组合：



5. 区间[1，3]外的数：



习题 4.2.2　将下面的 Scheme 函数转换为数轴上的区间：

```
1. (define (in-interval-1? x)
     (and (< -3 x) (< x 0)))

2. (define (in-interval-2? x)
     (or (< x 1) (> x 2)))

3. (define (in-interval-3? x)
     (not (and (<= 1 x) (<= x 5))))
```

阐明上述 3 个函数的合约与用途说明。手工计算下列表达式：

```
1. (in-interval-1? -2)
2. (in-interval-2? -2)
3. (in-interval-3? -2)
```

给出计算过程中较重要的中间结果，用图示的方法检查你的结果。

习题 4.2.3　包含一个变量的数学等式是关于一个未知数的断言。如二次方程

$$x^2 + 2 \cdot x + 1 = 0$$

是关于未知数 x 的一个断言。对于 $x=-1$，该断言成立：

$$x^2+2\cdot x+1=(-1)^2+2\cdot(-1)+1=1-2+1=0。$$

对于 $x=1$，该断言不成立：

$$x^2+2\cdot x+1=(1)^2+2\cdot(1)+1=1+2+1=4。$$

我们一般称使断言成立的值为方程的解。

等式可以表示为 Scheme 中的函数。如果有人说得到方程的一个解，我们可以使用该函数来验证该“解”是否确实是方程的解。上述例子相应的函数为：

```
;; equation1 : number -> boolean
;; 确定 x 是否是方程 x²+2·x + 1=0 的解
(define (equation1 x)
  (= (+ (* x x) (+ (* 2 x) 1)) 0))
```

对其个数值应用 *equation1* 的结果不是 true 就是 false，如

```
  (equation1 -1)
= true
```

而

```
  (equation1 +1)
= false
```

将下述等式转换为 Scheme 函数：

1. $4n + 2 = 62$;
2. $2n^2 = 102$;
3. $4n^2 + 6 \cdot n + 2 = 462$。

确定 10、12 或 14 是否是等式的解。

习题 4.2.4 等式不仅仅在数学中普遍存在，在程序设计中也经常使用。我们可以用等式说明一个函数对于输入应该如何计算，可以用等式说明手工计算表达式的过程，也可以将其添加在 DrScheme 的 Definitions 窗口中以测试程序例子。例如，如果程序的目标是定义函数 *Fahrenheit ->Celsius*，则可以在 DrScheme 的 Definitions 窗口中以如下方式添加程序例子：

```
;; 测试表达式:
(Fahrenheit->Celsius 32)
;; 预期值:
0
```

和

```
;; 测试表达式:
(Fahrenheit->Celsius 212)
;; 预期值:
100
```

按下 Exeoute 按钮执行程序，并对计算所得的结果和预期值进行比较。如果相等，则可以知道函数工作正常。

当计算结果变得越来越复杂时，对结果进行比较就会变得越来越乏味。实际上，使用“=”可以将上述等式转换为断言：

```
(= (Fahrenheit->Celsius 32)
   0)
```

和

```
(= (Fahrenheit->Celsius 212)
   100)
```

现在，如果所有断言的计算结果都为 true，则对于这些例子，程序工作正常。如果出现了 false，则说明程序仍然有错。

使用断言重新表示习题 2.2.1、2.2.2、2.2.3 和 2.2.4 中的例子。

测试：要设计出自动测试过程，需要对等式有更多的了解，将测试写成断言不失为一种好的经验。17.8 节将继续讨论等式和测试间的关系。

4.3　条件和条件函数

一些银行对于不同的存款给予不同的利率，客户存得越多，银行给的利率就越高。在这种情况下，利率依赖于存款额。为了帮助银行职员，银行使用一个利率计算函数，根据客户的存款额，函数将给出相应的利率。

利率计算函数必须确定对于给定的输入，哪个条件为真，因此说利率计算函数是一个条件函数。我们可以使用条件表达式来定义这些利率计算函数，条件表达式的一般形式为：

```
(cond                          (cond
  [question answer]              [question answer]
  ...                    或      ...
  [question answer])             [else answer])
```

其中省略号表示一个 cond 表达式可包含任意数目的 cond 行。每一 cond 行，也称为 cond 子句，它包含两个表达式，分别称为条件表达式(condition)和答案表达式(answer)。条件表达式是一个含有参数的布尔表达式，而答案表达式则是一个普通的 Scheme 表达式，若条件表达式为真，后者会根据参数的值来进行计算[1]。

到目前为止，条件表达式是我们所遇到的和将遇到的最复杂的表达式，因此，使用它们编写程序最容易产生错误，试比较下面两个表达式

```
(cond                          (cond
  [(< n 10) 5.0]                 [(< n 10) 30 12]
  [(< n 20) 5]                   [(> n 25) false]
  [(< n 30) true])               [(> n 20) 0])
```

左边是一个有效的 cond 表达式，因为每一 cond 行都包含了两个表达式。相反，右边就不是一个有效的 cond 表达式。因为第 1 行包含了 3 个而不是 2 个表达式。

计算 cond 表达式时，Scheme 先确定每个条件表达式的值，是 true 还是 false。对于第一个条件为 true 的 cond 子句，Scheme 执行其答案部分，答案部分的值就是整个 cond 表达式的值。若所有条件都为 false，而最后一个条件是 else，则 cond 表达式的值就是最后一个答案表达式的值。[2]

---

[1] 括号[和]是可选的，它们将不同的条件子句分隔开来，方便阅读函数。

[2] 如果 cond 表达式不包含 else 子句，并且所有条件的计算结果为 false，而 DrScheme 被设置为 Beginning Student 环境时，系统将给出一条错误消息。

以下两个简单的例子：

```
(cond
  [(<= n 1000) .040]
  [(<= n 5000) .045]
  [(<= n 10000) .055]
  [(> n 10000) .060])
```

```
(cond
  [(<= n 1000) .040]
  [(<= n 5000) .045]
  [(<= n 10000) .055]
  [else .060])
```

如果将 *n* 替换为 20000，则两个例子的前 3 个条件的计算结果皆为 false。而左边条件语句中的表达式(> 20000 10000)的计算结果为 true，因此答案为 0.60；对于右边的条件语句，else 子句给出了整个表达式的值，也是 0.60。反之，如果 *n* 为 1000，则值为.055，因为对于两个表达式来说，(<= 10000 1000)和(<= 10000 5000)的计算结果都为 false，而(<= 10000 10000)的计算结果为 true。

习题

习题 4.3.1 试判别下面两个表达式哪一个合法：

```
(cond
  [(< n 10) 20]
  [(>n 20) 0]
   [else 1])
```

```
(cond
  [(< n 10) 20]
  [(and(>n 20)(<=n 30))]
   [else 1])
```

对于不合法的表达式，试解释为何不合法。另外，下述条件表达式为何不合法？

```
(cond [(< n 10) 20]
      [* 10 n]
      [else 555])
```

习题 4.3.2 试确定当 *n* 为(a)500、(b)2800 和(c)15000 时下述表达式的值。

```
(cond
  [(<= n 1000) .040]
  [(<= n 5000) .045]
  [(<= n 10000) .055]
  [(> n 10000) .060])
```

习题 4.3.3 试确定当 *n* 为(a)500、(b)2800 和(c)15000 时下述表达式的值。

```
(cond
  [(<= n 1000)(* .040 1000)]
  [(<= n 5000)(+ (* 1000 .040)
                 (* (- n 1000) .045))]
  [else(+ (* 1000 .040)
          (* 4000 .045)
          (* (- n 10000) .055))])
```

借助 cond 表达式，现在可以定义本节开始时提到的利率计算函数。假定存款额小于等于 1000 美元的银行利率定为 4%。大于 1000 美元、小于等于 5000 美元定为 4.5%，大于 5000 美元定为 5%。显然，函数的输入为一个数值，而结果为另一个数值：

```
;; interest-rate : number -> number
```

```
;; interest-rate : number -> number
;; 确定给定 amount 存款额的利率
(define (interest-rate amount) ...)
```

而且，问题表述提供了 3 个例子：

```
(= (interest-rate 1000) .040)
(= (interest-rate 5000) .045)
(= (interest-rate 8000) .050)
```

注意，如果可能的话，例子将用布尔表达式表示。

函数主体应该是一个 **cond** 表达式，由它区分问题表述中所涉及的 3 种情况，以下是程序框架：

```
(cond
  [(<= amount 1000) ...]
  [(<= amount 5000) ...]
  [(> amount 5000) ...])
```

使用例子和上述框架，容易给出如下定义：

```
(define (interest-rate amount)
  (cond
    [(<= amount 1000) 0.040]
    [(<= amount 5000) 0.045]
    [(> amount 5000) 0.050]))
```

由于仅需考虑 3 种情况，还可以将第 3 个条件用 **else** 代替：

```
(define (interest-rate amount)
  (cond
    [(<= amount 1000) 0.040]
    [(<= amount 5000) 0.045]
    [else 0.050]))
```

对于某顾客的存款额（如 4000）当应用 *interest-rate* 时，计算过程会如预料的那样进行。Scheme 首先拷贝该函数主体，然后用 4000 代换 *amount*：

```
  (interest-rate 4000)
= (cond
    [(<= 4000 1000) 0.040]
    [(<= 4000 5000) 0.045]
    [else 0.050])
= 0.045
```

因为第一个条件的值为 false，而第 2 个条件的值为 true，因此程序的结果为 0.045 或 4.5%。如果使用(>*amount* 5000)而不是使用 else，计算过程也是一样的。

4.4　条件函数的设计

与设计一般函数相比，条件函数的设计比较复杂，程序设计者必须了解问题表述中所列出的不同情况并加以识别。为了强调这种思想的重要性，这里介绍并讨论条件函数的设计过程。该过程引入了一个

新的设计诀窍数据分析（data analysis），它要求程序设计者理解问题表述中所涉及的不同情况。因此，有必要对 2.5 节中所讨论的程序设计诀窍中的例子和程序体部分进行一些修改。

数据分析和定义：了解了问题表述所涉及的不同情况后，必须确定它们的数据定义 data definition，下面对这个思想进行深入讨论。

对于数学函数，一种好的策略是画出数轴，然后针对不同的情况确定相应的区间。考虑 *interest-rate* 函数的合约：

```
;; interest-rate : number -> number
;; 确定相应于存款额 amount(大于等于零)的利率
(define (interest-rate amount) ...)
```

该函数的输入是一个非负数，程序对于 3 种不同的情况给出不同的答案：



对于处理布尔值的函数，cond 表达式必须区分两种不同情况，即 true 和 false。我们很快将遇到其他形式的数据，这些数据需要对更多不同的情况进行推理。

函数例子：选择的例子应能说明不同的情况。如果这些情况可以用数值区间来刻划，还应该考虑所有的边界。

对于 *interest-rate* 函数，应该使用 0、1000 和 5000 作为例子。另外，也应选择 500、2000 和 7000 作为例子来检查区间内部数值的计算。

主体一条件：函数主体包含的 cond 表达式的数目应该与不同情况的数目一样，该要求提示了以下程序框架：

```
(define (interest-rate amount)
  (cond
    [... ...]
    [... ...]
    [... ...]))
```

接着必须阐明与每种情况相关的条件，条件是关于函数参数的断言，可以使用 Scheme 关系表达式或自定义的函数来表示。

对我们的例子而言，数轴区间的转换结果是如下 3 个表达式：

```
(and (<= 0 amount) (<= amount 1000))
(and (< 1000 amount) (<= amount 5000))
(< 5000 amount)
```

将它们加进函数，其最终结果为：

```
(define (interest-rate amount)
  (cond
    [(and (<= 0 amount) (<= amount 1000)) ...]
    [(and (< 1000 amount) (<= amount 5000)) ...]
    [(> amount 5000) ...]))
```

此阶段，程序设计者应该检查一下所选择的条件是否对输入进行了正确的区分，尤其是，如果一个输入值属于某一种特定情况并用 cond 子句进行了表示，那么位于该子句前的所有条件的计算结果都应该为 false，而该子句的条件的计算结果为 true。

主体一答案：最后，要确定对于每一个 cond 子句，函数应产生什么结果。具体地说，就是对于 cond 表达式中的每一行，如果条件为真，相应的表达式结果应该是什么。

对于我们的例子，结果由问题表述规定，分别是 4.0、4.5 和 5.0。对于更复杂的例子，必须按照第 1 个设计诀窍中的建议，对每个 cond 子句，确定一个表达式。

提示：如果 cond 的答案部分较为复杂，最好每次设计一个答案。假定条件的计算结果为 true，使用参数、基本运算和其他函数编写相应的答案，然后将整个函数应用于使条件为 true 的输入，并对所设计的答案进行计算，此时其他答案响仍可以保留为 "..."。

简化：完成了条件表达式的定义并对其进行测试之后，有些程序设计者仍然希望检查一下表达式条件是否能简化。在我们的例子中，由于 *amount* 的值总是大于等于 0，因此第一个条件表达式可以写成：

```
(<= amount 1000)
```

而且，cond 表达式是按顺序进行计算的。在对第 2 个表达式进行计算的时候，第 1 个表达式的计算结果肯定为 false，即 *amount* 的值不会小于等于 1000，这使得第 2 个条件的左边成分变得多余。经过进一步简化的 *interest-rate* 的程序框架为：

```
(define (interest-rate amount)
  (cond
    [(<= amount 1000) ...]
    [(<= amount 5000) ...]
    [(> amount 5000) ...]))
```

阶段	目标	任务
数据分析	确定函数所要处理的所有不同情况	检查问题表述理出不同的情况 • 枚举出所有可能的情况
例子	对于每种情况提供一个例子	对每种情况至少选择一个例子 • 对于区间或枚举值，例子必须包括边界
主体 (1) 条件	阐明一个条件表达式	写出 cond 表达式的框架，每种情况一个子句 • 对于每种情况阐明一个条件； • 确认条件能将例子适当区分
主体 (2) 答案	对于每个 cond 子句阐明条件答案	分别处理每个 cond 表达式 • 假定条件为真，设计相应的 Scheme 表达式，即条件答案

图 4.1　条件函数的主体的设计(使用图 2.2 中的设计诀窍)

图 4.1 总结了设计条件函数的一些建议，请与图 2.2 联系在一起阅读，并比较程序体的设计过程。请在设计一个条件函数之后再次阅读图 4.1。

习题

习题 4.4.1　设计函数 *interest*。与 *interest-rate* 类似，函数的输入为存款额。不同的是，计算结果是实际的年存款收益。假定银行规定存款额小于等于 1000 元时，利率为 4%；小于等于 5000 元时，利率为 4.5%；大于 5000 元时，利率为 5%。

习题 4.4.2　设计函数 *tax*，输入为雇员的毛收入，输出为应付的税款。税率计算方法如下，毛收入小于等于 240 美元的，税率为 0%，毛收入为 240 至 480 美元的，税率为 15%，毛收入在 480 美元以上的税率为 28%。

设计函数 *netpay*，其按雇员的每周工作时数计算其净收入。净收入为毛收入减去应付的税款。假定工作报酬为每小时 12 美元。

提示：当一个函数的定义变得太大或难以管理时，请使用辅助函数。

习题 4.4.3　一些信用卡公司对顾客一年的总消费额会给出一小部分奖赏，其中某一公司的奖赏表为：

1．最初 500 美元消费的奖赏为.25%；

2．接着的 1000 美元，即 500 美元到 1500 美元，消费的奖励为.50%；

3．接着的 1000 美元，即 1500 美元到 2500 美元，消费的奖励为.75%；

4．多于 2500 美元以上的奖励率为 1.0% 。

由此，若一个顾客一年使用信用卡的总消费额为 400 美元，其奖励金额就是于 0.25・1/100・400，即 1 美元；而消费 1400 美元的奖励金额为 5.75 美元，其等于前 500 美元消费的奖励金额 0.25・1/100・500 =1.25，加上另外 900 美元的奖励金额 0.50・1/100・900 = 4.50 美元。分别手工计算消费为 2000 美元和 2600 美元时的奖励金额。定义函数 *pay-back*，输入为一年的消费额，结果为相应的奖励金额。

习题 4.4.4　等式是关于数的断言，二次等式是一种特殊的等式，它的一般形式为：

$$a \cdot x^2 + b \cdot x + c = 0$$

其中参数 a、b 和 c 可以被替换为任意的数值，如

$$2 \cdot x^2 + 4 \cdot x + 2 = 0$$

或

$$1 \cdot x^2 + 0 \cdot x + (-1) = 0$$

其中变量 x 表示一个未知数。

等式两边计算结果是否相等依赖于 x 的取值(参见习题 4.2.3)。如果等式两边相等，称断言为真，否则称断言为假。使断言为真的数就是等式的解，容易看出，第 1 个等式有一个解，即−1：

$$2(-1)^2 + 4(-1) + 2 = 2 - 4 + 2 = 0$$

第 2 个等式有两个解：+1 和−1。

一个等式解的数目依赖于 a、b 和 c 的值。如果 a 的值为 0，这时相应的等式为一个退化方程，不再考虑它有多少个解。假定 a 不等于 0，则方程有

1．两个解，若 $b^2 > 4 \cdot a \cdot c$；

2．一个解，若 $b^2 = 4 \cdot a \cdot c$；

3．没有解，若 $b^2 < 4 \cdot a \cdot c$。

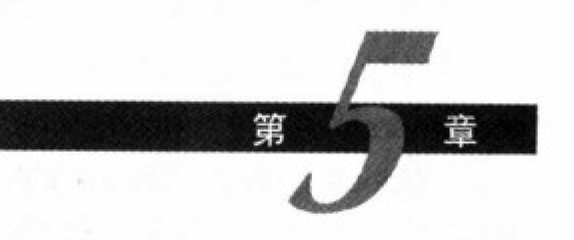
为了将最后一种情况和退化方程相区分，一般称非退化的等式为正则二次方程。

试编写函数 *how-many*，输入参数为一个二次方程的系数，即 a、b 和 c，该函数确定解的数目，如

```
(how-many 1 0 -1) = 2
(how-many 2 4 2) = 1
```

请给出更多的例子。对于每个例子，先手工确定方程解的数目，再使用 DrScheme 进行计算。如果方程不是正则的，请问函数要如何修改。

符号信息



今天计算机处理的大多是如同名字、词语和图像等符号类信息。Scheme 支持多种符号信息，包括符号(symbol)、字符串(string)、字符(character)和图像(images)等。符号是前面带一个单引号的一个键盘字符序列[1]：

```
'the  'dog  'ate  'a  'chocolate  'cat!  'two^3  'and%so%on?
```

与数值一样，符号并没有内在的含义，而由函数使用者将符号和现实世界的对象联系在一起，这种联系有时在特定的环境下是显而易见的，如'east 通常用来表示方向，即太阳升起的地方，而'professor 则表示大学中的一个教授。

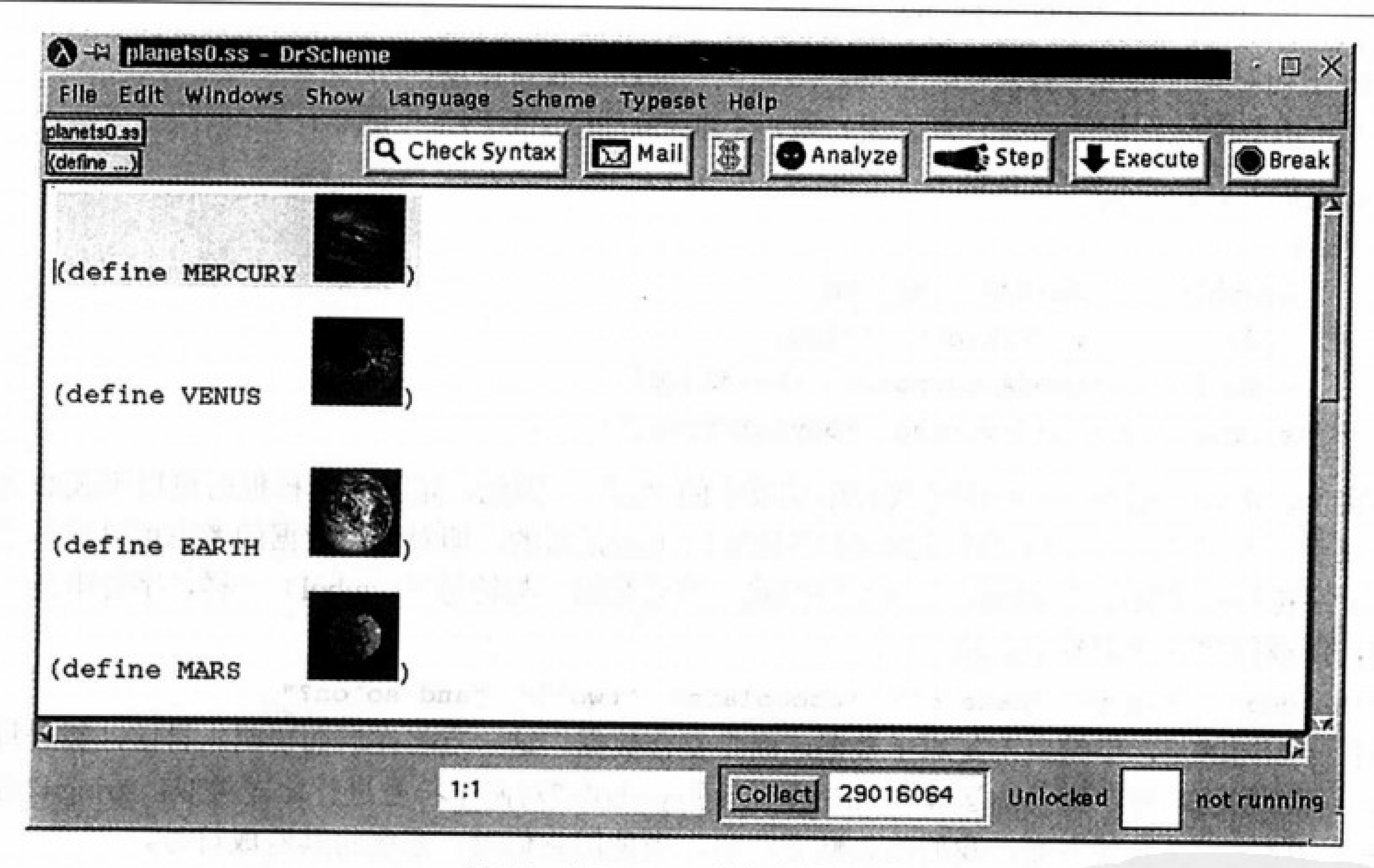


图 5.1　在 DrScheme 中定义行星图像

与数一样，符号是最基本的数据，它们用于表示如家庭、名字、头衔、命令、通知等信息。对于符号，Scheme 只提供一种操作：比较操作，即 symbol=?，它有两个参数，当且仅当它们相等时，其值为 true：

```
(symbol=? 'Hello 'Hello) = true
(symbol=? 'Hello 'Howdy) = false
(symbol=? 'Hello x) = true  如果 x 的值为 'Hello
```

[1] 并不是所有的键盘字符都是合法的符号，如空格和逗号就是不合法的。

```
(symbol=? 'Hello x) = false 如果x的值为 'Howdy
```

符号最早由人工智能研究者引入，用于设计能与人类进行交谈的函数。考虑函数 *reply*，它对问候"good morning"、"how are you"、"good afternoon"和"good evening"作出回答。这些问候语可以表示为'GoodMorning、'HowAreYou、'GoodAfternoon 和'GoodEvening。因此，*reply* 函数接受一个符号类型的参数，而结果也是符号类型的：

```
;; reply : symbol -> symbol
;; 对于问候 s 确定一个回答
(define (reply s) ...)
```

而且，函数必须区分 4 种情况，这意味着，按照 4.4 节所描述的设计诀窍，函数是一个包含 4 个子句的 **cond** 表达式：

```
(define (reply s)
  (cond
    [(symbol=? s 'GoodMorning) ...]
    [(symbol=? s 'HowAreYou?) ...]
    [(symbol=? s 'GoodAfternoon) ...]
    [(symbol=? s 'GoodEvening) ...]))
```

cond 子句对 4 个符号进行匹配，自然，这比区间的匹配容易得多。

从上述函数模板到最终函数只有一步之遥。下面是 *reply* 函数的一个版本：

```
(define (reply s)
  (cond
    [(symbol=? s 'GoodMorning) 'Hi]
    [(symbol=? s 'HowAreYou?) 'Fine]
    [(symbol=? s 'GoodAfternoon) 'INeedANap]
    [(symbol=? s 'GoodEvening) 'BoyAmITired]))
```

事实上，可以使用不同的回应替代程序模板中的"..."。因此，定义基本模板时可以不关心函数的输出。在下面的章节中，我们可以看到这种考虑实际上是正常的，即对输出数据的考虑可以推后进行。

关于字符串：字符串（string）是第 2 种形式的符号数据。与符号（symbol）一样，字符串是一个字符序列，但被包含在双引号中，如

```
"the dog"  "isn't"  "made of"  "chocolate"  "two^3"  "and so on?"
```

与符号不同的是，字符串不是原子数据，而是复合数据，这一点将在后面说明。目前，暂且将字符串看成一种特别的符号，惟一的运算是 string=?，同 symbol=?对两个符号进行比较类似，string=?对两个字符串进行比较。在其他方面，我们将忽略字符串，当使用它们时，系统将其看成符号。

关于图像：图像（image）是第 3 种形式的符号数据，开发能处理图像的函数是有趣的。与符号一样，图像本身并没有固有的含义，但我们往往趋向于将其和相关的信息联系在一起。

DrScheme 也支持图像。图 5.1 所示是一个对行星图像进行处理的函数。与数、布尔值一样，图像也可以出现在表达式中。通常，我们也给图像命名，因为它们一般被多个函数使用。如果你不喜欢某一个图像，可以简单地将其替换为另一个图像(参见 3.2 节)。

符号的手工练习

习题

习题 5.1.1　手工计算(*reply* 'HowAreYou?)，并与使用 DrScheme 按步执行功能所得的结果进行比较。使用 reply 将一完全的例子集合简洁地表示为布尔表达式（使用 symbol=? ）。

习题 5.1.2 设计函数 *check-guess*。输入为两个数，即 *guess* 和 *target*，根据它们的关系，函数产生输出'TooSmall、'Perfect 或'TooLarge 三者之一。事实上，该函数所实现的是二人猜数字游戏的一部分，在该游戏中，一方在 0 和 99999 间随机挑选一个数，而另一方确定该数是那一个数。对于每次猜测，前者的回答就是 *check-guess* 函数所给出的三种结果之一。

函数 *check-guess* 和教学软件包 guess.ss 一起实现了猜数字游戏的一方。教学软件包随机选择一个数据并在计算机屏幕弹出一个窗口，用户可以在窗口中选择一个数据并提交，所提交的数据由 check-guess 进行检查。游戏时，请将教学软件包中的 Languagel Set teachpack 选项设置为 guess.ss。在对函数 *check-guess* 进行彻底检查之后，使用下述表达式进行计算

```
(guess-with-gui check-guess)
```

习题 5.1.3 设计函数 *check-guess3*，与习题 5.1.2 不同的是，该函数处理的是用户逐一输入的数，而不是最后形成的数值。

为了简化问题，游戏仅使用 3 个数，因此 *check-guess3* 的输入为 3 个数和一个数 *target*，其中第 1 个数是个位数，第 2 个数是十位数，第 3 个数是百位数，而 *target* 是一个随机选择的数值。*check-guess3* 按照由 3 个数所确定的数与 *target* 的关系，产生下述答案之一：'TooSmall、'Perfect 或'TooLarge。游戏的其他部分仍然由 guess.ss 实现。欲进行 *check-guess3* 游戏，在对函数进行完全测试之后计算

```
(guess-with-gui-3 check-guess3)
```

提示：对于每个概念设计一个辅助函数。

习题 5.1.4 设计函数 what-kind，它的参数为一个二次方程的系数 a、b、c，该函数先确定方程是否退化，如果不是，再确定方程有多少个根，因此函数产生下列 4 个符号之一：'degenerate、'two、'one 或'none。

提示：参阅习题 4.4.4。

习题 5.1.5 设计函数 *check-color*，它是猜色游戏的主要部分，游戏参与者之一给两个方块挑选了两个颜色，它们是游戏的两个目标，游戏的另一个参与者猜测每个方块的颜色，第一个参与者对猜测给出下面四种可能的回答：

1. 'Perfect，如果第一个目标与第一个猜测相符合，并且第二个目标与第二个猜测相符合；
2. 'OneColorAtCorrectPosition，如果第一个目标与第一个猜测相符合或第二个目标与第二个猜测相符合；
3. 'OneColorOccurs，猜测的颜色在某一方块出现；
4. 'NothingCorrect，其他。

游戏的第一个参与者的回答只能是上述 4 个答案之一。第二个参与者的目标是用尽可能少的次数猜出方块的颜色。

函数 *check-color* 模仿第一个参与者的行为，它的参数是 4 种颜色，为简单起见，假定颜色的类型是符号，如'red，前两个参数是目标，后两个参数是猜测，函数的结果是上述 4 个答案之一。

在对函数进行测试之后，使用教学软件包中的 master.ss 进行游戏[1]，即计算(*master check-color*)并使用鼠标挑选颜色。

[1] 其操作方式不同于本游戏的商业版本 Master Mind.

第6章

复合数据之一：结构体

函数的输入很少局限于单一的度量（数值）、单一的开关位置（布尔值）或单一的名字（符号）。函数处理的数据通常是一个具有多个属性的对象，其中每个属性表示一种信息。例如，一个函数的输入可能是关于一张 CD 的记录，相关的信息可能包括艺术家的姓名、CD 的标题和 CD 的价格。类似地，如果要使用函数来刻划平面上一个物体的运动，则必须表示物体在平面上的位置、每个方向上的速度，可能的话，还有物体的颜色，等等。在这两种情况下，谈到几种信息时就好像它们是一个对象：一个记录或平面上的一个点。简而言之，可以将几种类型的数据组合为一种数据。

Scheme 提供了多种不同的数据组合方法。本章讨论结构体，结构体将固定数目的值组合为单一数据。第 9 章将讨论把任意数目的数据组合为单一数据的方法。

6.1 结构体

假定要在计算机屏幕上表示像素，像素类似于笛卡儿点，它有一个 x 坐标，表示像素在水平方向上的位置，有一个 y 坐标，表示像素在垂直方向上的位置，给定两个数值，就可以确定屏幕上的一个像素。

在 DrScheme 的教学软件包中，像素是一个 *posn* 结构体，包含两个数值，也可以说 *posn* 结构体是包含两个数的一个整体。可以使用操作 make-posn 创建一个 *posn* 结构体，该操作的输入是两个数值，输出是类型为 *posn* 的一个结构体，如：

```
(make-posn 3 4)
(make-posn 8 6)
(make-posn 5 12)
```

皆是 *posn* 结构体。与数相同，基本操作和函数都可以读入结构体，并返回结构体。

考虑计算给定像素和原点间距离的函数，函数的合约、头部和用途说明可以简单地阐述为：

```
;; distance-to-0 : posn -> number
;; 计算一个 posn 和原点的距离
(define (distance-to-0 a-posn) ...)
```

可以看出 *distance-to-0* 的输入为一个简单的值、一个 *posn* 结构体，结果也是一个简单的值，即数。

对于例子，输入可以是上面提到的 3 个 *posn* 结构体，目前所需要的是输入与输出相联系的例子。显而易见，如果 **0** 是坐标之一，则函数结果就是另一坐标值：

```
  (distance-to-0 (make-posn 0 5))
= 5
```

和

```
  (distance-to-0 (make-posn 7 0))
= 7
```

一般来说，由几何学原理，坐标为 x 和 y 的点离原点的距离为

$$\sqrt{x^2 + y^2}。$$

因此，

```
  (distance-to-0 (make-posn 3 4))
= 5

  (distance-to-0 (make-posn 8 6))
= 10

  (distance-to-0 (make-posn 5 12))
= 13
```

现在，将注意力转向函数的定义。虽然例子说明 *distance-to-0* 的设计不需要区分不同的情况。但我们还是束手无策，因为 *distance-to-0* 的单个参数表示的是整个像素，而计算距离却需要两个坐标的值。从另一个角度来说，我们知道如何使用 make-posn 将两个数值组合为一个 *posn* 结构体，但不知道如何从一个 *posn* 结构体中提取这些数值。

幸运的是，Scheme 提供了从结构体中提取值的操作[1]。对于 *posn* 结构体，有两个操作：posn-x 和 posn-y，前者提取 x 坐标，后者提取 y 坐标。

下述等式描述了 posn-x、posn-y 和 make-posn 之间的关系：

```
  (posn-x (make-posn 7 0))
= 7
```

和

```
  (posn-y (make-posn 7 0))
= 0
```

等式说明了已知的事实，假定有以下定义：

```
  (define a-posn (make-posn 7 0))
```

那么就可以在 DrScheme 的 Interactions 窗口中进行如下运算：

```
  (posn-x a-posn)
= 7

  (posn-y a-posn)
= 0
```

自然，我们也可以嵌套使用此类表达式：

```
  (* (posn-x a-posn) 7)
= 49

  (+ (posn-y a-posn) 13)
= 13
```

现在已经有了定义 *distance-to-0* 所需要的所有知识：函数的 *a-posn* 参数是一个 *posn* 结构体，该结构

[1] 另一个术语是“访问一个记录的域”。我们可以将结构值看成一个容器，从中我们可以得到其他值。

体包含了两个数值，可以使用(**posn-x** *a-posn*)和(**posn-y** *a-posn*)提取它们。将这些知识加到函数定义框架之中，有：

```
(define (distance-to-0 a-posn)
  ... (posn-x a-posn) ...
  ... (posn-y a-posn) ...)
```

使用框架和例子，函数其余部分的定义就容易了：

```
(define (distance-to-0 a-posn)
  (sqrt
    (+(sqr (posn-x a-posn))
       (sqr (posn-y a-posn)))))
```

函数先分别计算(**posn-x** *a-posn*)和(**posn-y** *a-posn*)，即坐标 *x* 和 *y*，再求平方和，最后求平方根。使用 DrScheme。 可以快速检验新函数的计算结果是正确的。

习题

习题 6.1.1 手工计算下列表达式：

```
1.(distance-to-0 (make-posn 3 4));
2.(distance-to-0 (make-posn (* 2 3) (* 2 4)));
3.(distance-to-0 (make-posn 12 (- 6 1)))。
```

假定 **sqr** 是单步执行的计算，请给出所有计算步骤，并将结果和使用 DrScheme 单步执行器所得的结果进行比较。

6.2 补充练习：绘制简单图形

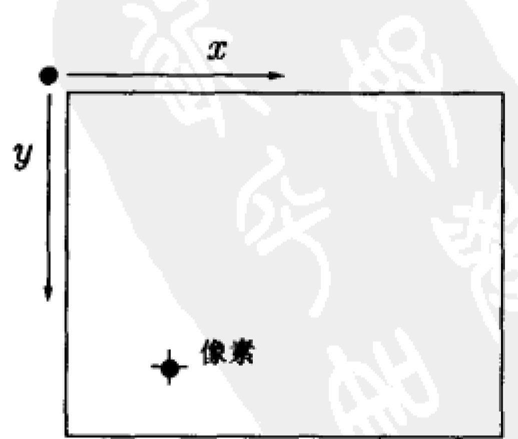
DrScheme 提供的图形教学软件包 draw.ss 包含如下绘图操作：

1. *draw-solid-line*，在画布上绘制直线，输入为 2 个 *posn* 结构体和 1 种颜色，其中 *posn* 结构体表示直线的始点和终点；

2. *draw-solid-rect*，在画布上绘制长方形，它读入 4 个参数，分别是表示长方形左上角位置的 *posn* 结构体，长方形的宽，长方形的高，以及边的颜色；

3. *draw-solid-disk*，在画布上绘制圆盘，它读入 3 参数，分别是表示圆盘中心的 *posn* 结构体，圆盘的半径以及颜色；

4. *draw-circle*，在画布上绘制圆，它读入 3 个参数，分别是表示圆心位置的 *posn* 结构体，圆的半径以及颜色。



如果能如预期的那样成功地改变画布的状态，上述操作的结果就为 true；如果操作错误，在计算过程停止的同时系统会给出错误消息。通常称画布上的操作结果为效果，本书第七部分将对效果进行较深入的研究。

上述绘图操作有相匹配的 *clear* 操作：*dear-solid-line*、*dear-solid-rect*、*dear-solid-disk* 以及 *dear-circle*，把 *clear* 操作作用于

与 *draw* 函数相同的参数，效果是清除画布上的相应的图形[1]。计算机绘图操作将屏幕解释为：

可以看出：一，平面上的原点位于左上角；二，y 坐标轴的方向朝下。理解上图和通常意义的笛卡儿平面对于正确绘制图形非常关键。

习题

习题 6.2.1　按顺序计算下列表达式：

1. (*start* **300 300**)：为后续绘图操作打开一个画布；
2. (*draw-solid-line* **(make-posn 1 1) (make-posn 5 5) 'red**)：绘制一条红线；
3. (*draw-solid-rect* **(make-posn 20 10) 50 200 'blue**)：绘制一个宽为 50，长为 200 的蓝色长方形；
4. (*draw-circle* **(make-posn 200 10) 50 'red**)：绘制一个半径为 50 的红色的圆，圆心在长方形上面一条边上；
5. (*draw-solid-disk* **(make-posn 200 10) 50 'green**)：绘制一个半径为 50 的绿色圆盘，圆心在长方形上面一条边上；
6. (*stop*)：关闭画布。

请点击 DrScheme 的 HelpDesk 菜单项，阅读 draw.ss 的文档。

图 6.1 中的定义和表达式的功能是绘制交通红绿灯。程序阐述了全局常量的定义的使用。在程序中，常量刻划了表示交通灯轮廓的画布幅度，以及三个灯泡的位置。

```
;; 红绿灯的大小
(define WIDTH 50)
(define HEIGHT 160)
(define BULB-RADIUS 20)
(define BULB-DISTANCE 10)

;; 灯泡的位置
(define X-BULBS (quotient WIDTH 2))
(define Y-RED (+ BULB-DISTANCE BULB-RADIUS))
(define Y-YELLOW (+ Y-RED BULB-DISTANCE (* 2 BULB-RADIUS)))
(define Y-GREEN (+ Y-YELLOW BULB-DISTANCE (* 2 BULB-RADIUS)))

;; 绘制红灯亮时的灯光
(start WIDTH HEIGHT)
(draw-solid-disk (make-posn X-BULBS Y-RED) BULB-RADIUS 'red)
(draw-circle (make-posn X-BULBS Y-YELLOW) BULB-RADIUS 'yellow)
(draw-circle (make-posn X-BULBS Y-GREEN) BULB-RADIUS 'green)
```

图 6.1　绘制红绿灯

习题 6.2.2　设计函数 *clear-bulb*，读入'green、'yellow 或'red 之一，输出为 true，效果是关掉红绿灯上相应颜色的灯，即清除相应颜色的圆盘并以相同颜色的圆代替。

设计诀窍选择：参见第 5 章，了解输入为枚举类型的函数的设计。

测试：在设计绘制图形的函数时，没有考虑到测试表达式。尽管可以编制一个测试软件包，但它超出了本书的范围。

效果组合：绘制和清除圆盘的操作在成功完成任务后所产生的结果都是 true。将这些值和效果组合在一起的自然方法是使用 and 表达式。特别地，如果 *exp1* 和 *exp2* 都产生效果，并且希望在 *exp1* 之后看到 *exp2* 的效果，可以写成

```
(and exp1 exp2)
```

[1] 更多信息，请参阅 Dr Scheme 的帮助。

后面将详细讨论效果的生成并学习将效果组合在一起的不同方式。

练习 6.2.3 设计函数 *draw-bulb*，输入为'green、'yellow 或'red 之一，输出为 true，效果是打开红绿灯上相应颜色的灯。

练习 6.2.4 设计函数 *switch*，输入是两个符号，分别代表红绿灯上的两种颜色，输出为 true，效果为先关闭第一个符号表示的灯，然后打开第二个符号表示的灯。

练习 6.2.5 下面是函数 *next*：

```
;; next : symbol -> symbol
;; 将当前红绿灯颜色转换为下一颜色
(define (next current-color)
  (cond
    [(and (symbol=? current-color 'red) (switch 'red 'green))
     'green]
    [(and (symbol=? current-color 'yellow) (switch 'yellow 'red))
     'red]
    [(and (symbol=? current-color 'green) (switch 'green 'yellow))
     'yellow]))
```

函数的输入为红绿灯的当前颜色，输出为红绿灯的下一个颜色。即如果输入为'green，输出为'yellow；输入为'yellow，输出为'red；输入为'red，输出为'green。

将图 6.1 的最后 3 行代码替换为(*draw-bulb* **red**)，使红绿灯的当前状态为红灯亮，然后使用 *next* 函数将红绿灯颜色进行 4 次转换。

6.3 结构体定义

上一节探讨了 *posn* 结构体，该结构体包含两个数值，可用于表示像素。如果要表示一个雇员的记录或者三维空间中的一个点，它就没有用武之地了。因此，DrScheme 允许程序设计者定义自己的结构体，用以表示属性数目固定的任何类型的对象。

结构体定义是一种新的定义形式。下面是 DrScheme 中 *posn* 的定义：

```
(define-struct posn (x y))
```

DrScheme 对该结构体定义进行计算的结果是创建 3 个操作，程序设计者可使用这些操作创建数据并在编程中使用：

1. **make-posn**：构造器，用于创建一个 *posn* 结构体；
2. **posn-x**：选择器，用于提取 *x* 坐标；
3. **posn-y**：选择器，用于提取 *y* 坐标。

通常，构造器的前缀为“make-”，选择器的后缀为字段名。这种命名规范看起来比较复杂，但稍加练习，便可轻松掌握。

考虑下列表达式：

(define-struct *entry* (*name zip phone*)**)**

其所定义的结构体是通讯录中的一个简化条目，每个条目包括 3 个值。或者说每个 *entry* 结构体有 3 个字段：*name*、*zip* 和 *phone*。因此构造器 make-entry 的输入有 3 个值，例如：

```
(make-entry 'PeterLee 15270 '606-7771)
```

创建了一个 *entry* 结构体，*name* 字段的值为'PeterLee，*zip* 字段的值为 15270，*phone* 字段的值为'606-7771。

可以把结构体看作为盒子，其中隔间的数目和字段的数目一样多，通过将值放进隔间，可以得到 *entry*

结构体的图解，如下：

```
Name:          zip:         phone:
'PeterLee      15270      606-7771
```

其中斜体标签是字段的名字。

在使用 **define-struct** 定义 *entry* 结构体的同时，系统也引入了 3 个新的选择器：

```
entry-name    entry-zip    entry-phone
```

以下是使用第 1 个选择器的例子：

```
  (entry-name (make-entry 'PeterLee 15270 '606-7771))
= 'PeterLee
```

如果给结构体一个名字：

```
(define phonebook (make-entry 'PeterLee 15270 '606-7771))
```

那么就可以在 Interaction 窗口使用选择器提取结构体任何一个字段中的数据，例如：

```
  (entry-name phonebook)
= 'PeterLee
  (entry-zip phonebook)
= 15270
  (entry-phone phonebook)
= '606-7771
```

形象点说，就是构造器创建一个带有多个隔间的盒子，并将值放置其中，选择器显示指定隔间存放的值，而不影响盒子。

最后一个例子表示的是摇滚歌星的信息，表达式

```
(define-struct star (last first instrument sales))
```

定义了结构体 *star*，有 4 个字段。相应地，有 5 个基本操作：make-star、star-last、star-first、star-instrument 和 star-sales。第 1 个操作用于构造 *star* 结构体，其他是从 *star* 结构体提取值的选择器操作。

创建 *star* 结构体的方法是将 **make-star** 应用于 3 个符号和 1 个正整数：

```
(make-star 'Friedman 'Dan 'ukelele 19004)
(make-star 'Talcott 'Carolyn 'banjo 80000)
(make-star 'Harper 'Robert 'bagpipe 27860)
```

要选择一个称为 *E* 的歌星结构体的名字，可使用

```
(star-first E)
```

类似地，使用其他选择器可提取其他字段的值。

习题

习题 6.3.1　考虑下列结构体定义：

```
1. (define-struct movie (title producer))
2. (define-struct boyfriend (name hair eyes phone))
3. (define-struct cheerleader (name number))
4. (define-struct CD (artist title price))
5. (define-struct sweater (material size producer))
```

对于每一个结构体定义，对应的 Scheme 构造器和选择器的名字是什么？请画出表示每个结构体的盒子。

习题 6.3.2　考虑下列结构体定义：

```
(define-struct movie (title producer))
```

并计算下列表达式：

1. (movie-title (make-movie 'ThePhantomMenace 'Lucas))；
2. (movie-producer (make-movie 'TheEmpireStrikesBack 'Lucas))。

假定 x 和 y 代表任意的符号，计算下列表达式：

1. (movie-title (make-movie x y))
2. (movie-producer (make-movie x y))

给出描述 movie-title、movie-producer 和 make-movie 间关系的等式。

函数的输入和输出都可以是结构体。假如要定义一个函数，记录某明星唱片的销售增量，函数输入为一个 *star* 结构体，输出也为一个 *star* 结构体，显而易见，这个输出的结构体除了销量值外，与输入结构体相同。现假定要将某明星的唱片销售量增加 20000 张。

先使用合约、头部和用途说明给出函数的基本描述：

```
;; increment-sales : star -> star
;; 将 star 的销量值增加 20000
(define (increment-sales a-star) ...)
```

下面这个例子说明函数是如何处理 *star* 结构体的：

```
(increment-sales (make-star 'Abba 'John 'vocals 12200))
```

结果应该是：

```
(make-star 'Abba 'John 'vocals 32200))
```

上面提到的 3 个 *star* 结构体也可以作为输入。

increment-sales 函数构造了一个新的 *star* 结构体 *make-star*，为了完成此任务，它必须从 *a-star* 中提取数据。事实上，几乎所有 *a-star* 的数据都是函数所产生的新结构体中的数据，这表明 *increment-sales* 的定义应该包括如下表达式，用来提取 *a-star* 中 4 个字段值：

```
(define (increment-sales a-star)
  ... (star-last a-star)...
  ... (star-first a-star)...
  ... (star-instrument a-star)...
  ... (star-sales a-star) ... )
```

正如在例子中所看到的那样，函数求取 20000 与(**star-sales** *a-star*)之和，再使用 make-star 将 4 个数据组合为一个 *star* 结构体。完整的函数定义如图 6.2 所示。

```
;; increment-sales : star -> star
;; 将 a-star 记录中销量的值增加 20000
(define (increment-sales a-star)
  (make-star (star-last a-star)
             (star-first a-star)
             (star-instrument a-star)
             (+ (star-sales a-star) 20000)))
```

图 6.2 *increment-sales* 函数的完整定义

习题

习题 6.3.3　给出表示喷气式飞机的结构体定义，假定飞机有 4 个基本属性：名称('f22、'tornado 或'mig22)、加速度、最高时速和航程。设计函数 *within-range*，输入为飞机记录和目标离开基地的距离，函数确定飞机是否可以到达指定目标。进一步开发函数 *reduce-range*，输入为飞机记录，输出也是飞机记录，但其中 *range* 字段的值为原始值的 80%。

6.4 数据定义

考虑下列表达式：

```
(make-posn 'Albert 'Meyer)
```

其构造一个包含两个符号的 *posn* 结构体。如果将函数 *distance-to-0* 应用于该结构体，计算将以失败告终：

```
  (distance-to-0 (make-posn 'Albert 'Meyer))

= (sqrt
    (+ (sqr (posn-x (make-posn 'Albert 'Meyer)))
       (sqr (posn-y (make-posn 'Albert 'Meyer)))))

= (sqrt
    (+ (sqr 'Albert)
       (sqr (posn-y (make-posn 'Albert 'Meyer)))))

= (sqrt
    (+ (* 'Albert 'Albert)
       (sqr (posn-y (make-posn 'Albert 'Meyer)))))
```

也就是说，表达式要求**'Albert** 与自身相乘，产生错误。类似地，

```
(make-star 'Albert 'Meyer 10000 'electric-organ)
```

不会产生一个 *star* 结构体，特别是，它不能被 *increment-sales* 所处理。

为了避免此类问题，Scheme 给每个结构体定义加上一个数据定义，它以自然语言和 Scheme 语言相混合的形式说明了程序设计者应该如何使用和构造此类结构体数据。例如，以下是 *posn* 结构体的数据定义：

> *posn* 是结构体：
>
> **(make-posn** *x y*)
>
> 其中 *x* 和 *y* 是数。

它说明一个合法的 *posn* 结构体包含两个数，而不是别的东西。因此，欲使用 make-posn 创建一个 *posn* 结构体，必须将构造函数应用于两个数，对于 *posn* 结构体使用选择器的结果是数。

Star 结构体的数据定义稍微有点复杂：

> *star* 是结构体：
>
> **(make-star** *last first instrument sales*)
>
> 其中 *last*、*first* 和 *instrument* 是符号，而 *sales* 是数。

这个数据定义说明，一个有效的 *star* 结构体的 *last*、*first* 和 *instrument* 字段的值皆为符号，而 *sales* 字段的值为数。

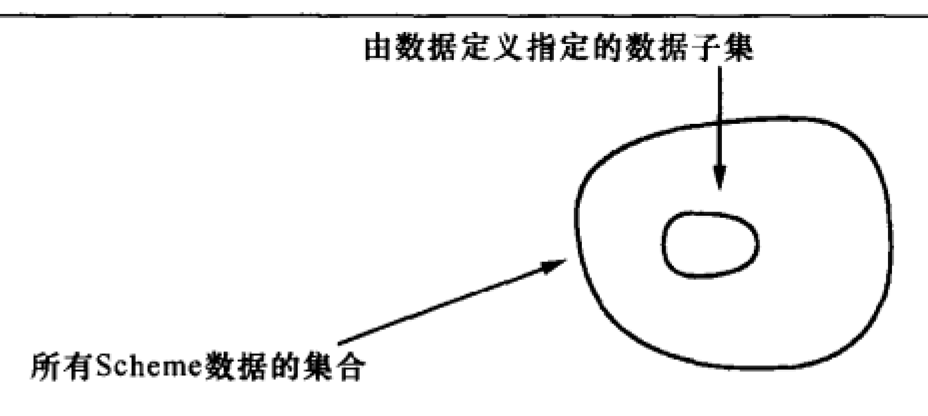


图 6.3　数据定义的含义

如图 6.3 所示，通常数据定义给出了 Scheme 数据集合的一个子类。Scheme 所有可能的取值包括数、符号、图像、字符、字符串、布尔值和其他结构体。而所定义的函数，仅仅是处理该集合的一个子集。例如，*area-of-disk* 的输入为数，第 5 章中的 *reply* 函数的输入为符号。一些子类如 *number*，由于对所有程序设计任务来说都是有用的，因此有一个名字，而其他的子类仅仅在一个特殊的环境下才是有意义的，对于这些情况，程序员应该引入数据定义对它们进行说明。

数据定义是程序设计者和用户间的界面，这是它最重要的作用。程序设计者和用户间都应该尊重数据定义，前者还应该在函数构造过程中使用它。例如，当 *distance-to-0* 的设计者规定所有 *posn* 结构体都包含两个数时，该函数的使用者必须将其应用于恰好包含两个数的 *posn* 结构体。程序设计者在函数开发过程中应该使用数据定义。当然，以自然语言和 Scheme 混合表示的数据定义并不能阻止滥用 make-posn 的可能性。数据定义事实上只是一份书面意图声明，但任何有意或无意违反该声明的人都将面临异常的计算结果[1]。

习题

习题 6.4.1　给出下列结构体定义的数据定义：

```
1. (define-struct movie (title producer))
2. (define-struct boyfriend (name hair eyes phone))
3. (define-struct cheerleader (name number))
4. (define-struct CD (artist title price))
5. (define-struct sweater (material size producer))
```

请合理地假定每个字段所属的数据类型。

习题 6.4.2　假定一个时刻由 3 个数组成：时、分、秒。请给出结构体定义和数据定义，用来表示从午夜开始计算的时刻。

练习 6.4.3　假定单词是用'a 到'z 之间的字符来表示，请给出由 3 个字母组成的单词的结构体定义和数据定义。

[1] 对于给定的结构体，DrScheme 包含了一个检查用户和程序设计者是否遵循数据定义的机制。此时，程序设计者必须以一种特殊的语言来表达数据定义。尽管对于大型程序来说，检查数据定义是否被遵循是非常重要的，但对导论性的课程来说，可以忽略这一点。

6.5　设计处理复合数据的函数

前几节采用常规方法来设计处理复合数据的函数。首先，程序程序设计者必须了解何时需要使用结构体，简单原则是，如果某个对象的描述需要若干种信息则使用结构体。如果不使用结构体，程序设计者可能很快会对哪些信息属于哪个对象失去线索，当编写处理大量数据的大型函数时，尤其如此。

其二，程序设计者可以使用结构体定义和数据定义来组织函数。在设计函数的时候可以使用模板，正如本节和以后章节中所看到的那样，模板和数据定义相匹配，这是函数设计中的基本要素。图 6.4 是一个完整的例子。

```
;; 数据分析和定义:
(define-struct student (last first teacher))
;; student 是结构体(make-student l f t)，其中 f、l 和 t 是符号

;; 合约：subst-teacher : student symbol -> student

;; 用途说明:如教师的名字为'Fritz，创建一个 student 结构体，把教师的名字替换为新的

;; 例子:
;; (subst-teacher (make-student 'Find 'Matthew 'Fritz) 'Elise)
;; =
;; (make-student 'Find 'Matthew 'Elise)
;; (subst-teacher (make-student 'Find 'Matthew 'Amanda) 'Elise)
;; =
;; (make-student 'Find 'Matthew 'Amanda)
;; 模板：
;; (define (process-student a-student a-teacher)
;; ... (student-last a-student) ...
;; ... (student-first a-student) ...
;; ... (student-teacher a-student) ...)
;; 定义:

(define (subst-teacher a-student a-teacher)
  (cond
    [(symbol=? (student-teacher a-student) 'Fritz)
     (make-student (student-last a-student)
                   (student-first a-student)
                   a-teacher)]
    [else a-student]))

;; 测试:
(subst-teacher (make-student 'Find 'Matthew 'Fritz) 'Elise)
;; 预期值:
(make-student 'Find 'Matthew 'Elise)

(subst-teacher (make-student 'Find 'Matthew 'Amanda) 'Elise)
;; 预期值:
(make-student 'Find 'Matthew 'Amanda)
```

图 6.4　复合数据的设计诀窍：一个完整的例子

为了强调这点，下面改写第 2.5 节中的设计诀窍，使之适合复合数据。重要的是，使用复合数据需

要对一些设计诀窍作出调整，我们需要两个新的步骤：数据分析和模板设计。

数据分析和设计：在开始函数设计之前，先必须了解如何在程序设计语言中表示问题的信息。为做到这一点，需要搜索问题以获得相关对象的描述，然后基于分析结果设计数据表示。

通常使用 Scheme 的原子数据（数、符号和图像等）来表示信息，如果发现一个对象有 N 个属性，可以引入一个有 N 个字段的结构体，同时给出每个字段的数据定义。

考虑处理学生记录的函数。如果学校所感兴趣的学生属性为：

1. 名；
2. 姓；
3. 班级教师的姓名。

则可以将学生的信息表示为下列结构体：

```
(define-struct student (last first teacher))
```

下面是数据定义，它尽可能精确地说明了 *student* 结构体：

> *student* 是结构体：
>
> **(make-student** *l f t***)**
>
> 其中 *l*、f 和 *t* 是符号。

该数据类型包含如下结构体：

```
(make-student 'findler 'kathi 'matthias)
(make-student 'fisler 'sean 'matthias)
(make-student 'flatt 'shriram 'matthias)
```

合约：为了阐述合约，可以使用诸如数和符号等原子类型的数据名称（如 *number* 和 *symbol*），以及在数据定义中引入的名字（如 *student*）。

模板：一个读入复合数据的函数在计算中一般会使用输入数据的组成成分。为此，先要设计一个模板。对于复合数据来说，模板包括函数头部和主体，主体则列出了所有可能的选择器表达式。

换句话说，模板表示了程序对于输入的了解，并且不涉及输出。因此，所有输入相同结构体的函数可以使用相同的模板。另外，由于模板与函数目的无关，可以在例子之前或之后进行阐述。

考虑输入为 *student* 结构体和教师名字的函数：

```
;; process-student : student symbol -> ???
(define (process-student a-student a-teacher) ...)
```

其中 *a-student* 是一个表示结构体的参数，*a-teacher* 则表示一个符号，模板如下：

```
;; process-student : student symbol -> ???
(define (process-student a-student a-teacher)
  ... (student-last a-student) ...
  ... (student-first a-student) ...
  ... (student-teacher a-student) ...)
```

其中 *???* 表示对函数的输出目前还没有给出任何假定。对任何以 *student* 结构体为输入的函数，都可以使用该模板。

例子：下面考虑两个读入 *student* 结构体的函数。第一个函数是 *check*，如果教师的名字等于 *a-teacher*，该函数返回学生的名字，否则返回'none：

```
(check (make-student 'Wilson 'Fritz 'Harper) 'Harper)
;; 预期值:
'Wilson
```

```
(check (make-student 'Wilson 'Fritz 'Lee) 'Harper)
;; 预期值：
'none
```

第二个函数是 *transfer*，在它返回的结构体中，*teacher* 字段的值为 *a-teacher*，除 *teacher* 字段以外，其余成分的值和 *a-student* 一样：

```
(transfer (make-student 'Wilson 'Fritz 'Harper) 'Lee)
;; 预期值：
(make-student 'Wilson 'Fritz 'Lee)

(transfer (make-student 'Woops 'Helen 'Flatt) 'Fisler)
;; 预期值：
(make-student 'Woops 'Helen 'Fisler)
```

主体：与上例一样，模板给函数的定义提供了诸多线索。这一步的目标是阐明一个表达式，它使用 Scheme 的基本操作及其他函数由已知数据计算出答案。由模板可知，已知数据是参数以及由选择器表达式表示的数据。要确定选择器表达式会输出什么，可以阅读结构体的数据定义。

我们回过头看第一个例子 check:

```
(define (check a-student a-teacher)
  (cond
    [(symbol=? (student-teacher a-student) a-teacher)
     (student-last a-student)]
    [else 'none]))
```

模板包含了 3 个选择器表达式，该函数使用了 2 个。函数对选择器表达式(**student-teacher** *a-student*)和 *a-teacher* 进行了比较，如果二者相等，产生结果(**student-last** *a-student*)。由选择器表达式结果的名字和问题的表述，函数的定义是显而易见的。

类似地，使用模板和例子可以容易地得到函数 *transfer* 的定义：

```
(define (transfer a-student a-teacher)
  (make-student (student-last a-student)
                (student-first a-student)
        a-teacher))
```

与第一个函数一样，这里也使用了两个选择器表达式，但是使用的方式不同。而重要的是，模板提示了所有可用的信息。定义函数时，可以使用和组合各种可用信息。

图 6.5 以表的形式给出了处理复合数据的函数设计诀窍。在实践中，一个函数可能包含许多其他函数，它们都对相同类型的输入数据进行处理。因此模板可以被重用多次，这也意味着例子的构造应该在模板设计之后。请将图 6.5 与图 2.2 和图 4.1 中的设计诀窍进行比较。

习题

习题 6.5.1　为输入是下述结构体的函数设计模板：

1. (**define-struct** *movie (title producer)*)
2. (**define-struct** *boyfriend (name hair eyes phone)*)
3. (**define-struct** *cheerleader (name number)*)
4. (**define-struct** *CD (artist title price)*)
5. (**define-struct** *sweater (material size producer)*)

习题 6.5.2 设计函数 *time->seconds*，它读入一个 **time** 结构体（参见练习 6.4.2），返回从午夜至 **time** 结构体所表示的时刻之间的秒数，例如：

```
(time->seconds (make-time 12 30 2))
;; 预期值:
45002
```

另外，请给出例子的解释。

阶　段	目　标	任　务
数据分析和设计	阐明数据定义	确定在问题表述中所涉及的对象的数据种类 • 对于每类数据增加结构体定义和数据定义
合约、用途说明和函数头头	给函数命名； 说明输入和输出数据的类型； 描述函数的用途说明； 阐明函数头部	给函数命名、说明输入数据的类型，输出数据的类型，指出函数的目的： ;; *name* : *in1 in2* ...--> *out* ;; 从 *x1* ..计算 ... (define (*name x1 x2* ...) ...)
例子	使用例子刻划输入和输出间的关系	搜索问题表述中的例子 • 计算例子； • 如果可能的话，验证结果； • 构造例子
模板	阐明程序框架	若参数是复合数据，使用选择器表达式填写主体 • 如果函数是条件式的，写出所有合适的分枝
主体	定义函数	使用 Scheme 基本操作、其他函数、选择器表达式和变量设计 Scheme 表达式
测试	发现错误（拼写错误和逻辑错误）	将函数应用于例子中的输入 • 检查程序输出是否与预期的值相符

图 6.5　设计处理复合数据的函数（图 2.2 中设计诀窍的精化）

6.6 补充练习：圆和长方形的移动

在设计计算机游戏的时候，通常要求在计算机屏幕上移动一个图像。例如，图 6.2 所示表示了一个简单的脸形图案从画布的左边到右边的移动。为简单起见，这里的图形仅包括长方形和圆。第 6.2 节中我们已经学习了如何绘制和删除一个圆。这里我们学习如何平移一个圆，使圆沿着一条直线移动。第 7.4、10.3 和 21.4 节将使用简洁的程序说明如何移动整个图片[1]。

遵循设计诀窍，我们先设计结构体定义和数据定义，然后是模板，最后再编写必需的函数。第一个系列练习涉及圆，第二个系列练习涉及长方形。

关于逐步求精：开发大型程序的方法之一是逐步求精。逐步求精的基本思想是从简单版本的程序出

[1] 这些章节受到了 Karen Buras 女士和她的儿子的启发。

发，即先处理问题中最重要的部分。本节先设计最简单的程序，即移动圆和长方形的程序，以后再对程序进行精化，使其可以处理越来越复杂的对象。例如，第 10.3 节将讨论如何对包含任意数目的圆和长方形的图形进行处理。一旦开发出完整的程序，再对其进行编辑，使其他人也可以方便地阅读和修改。第 21.4 节将讨论这方面的内容。

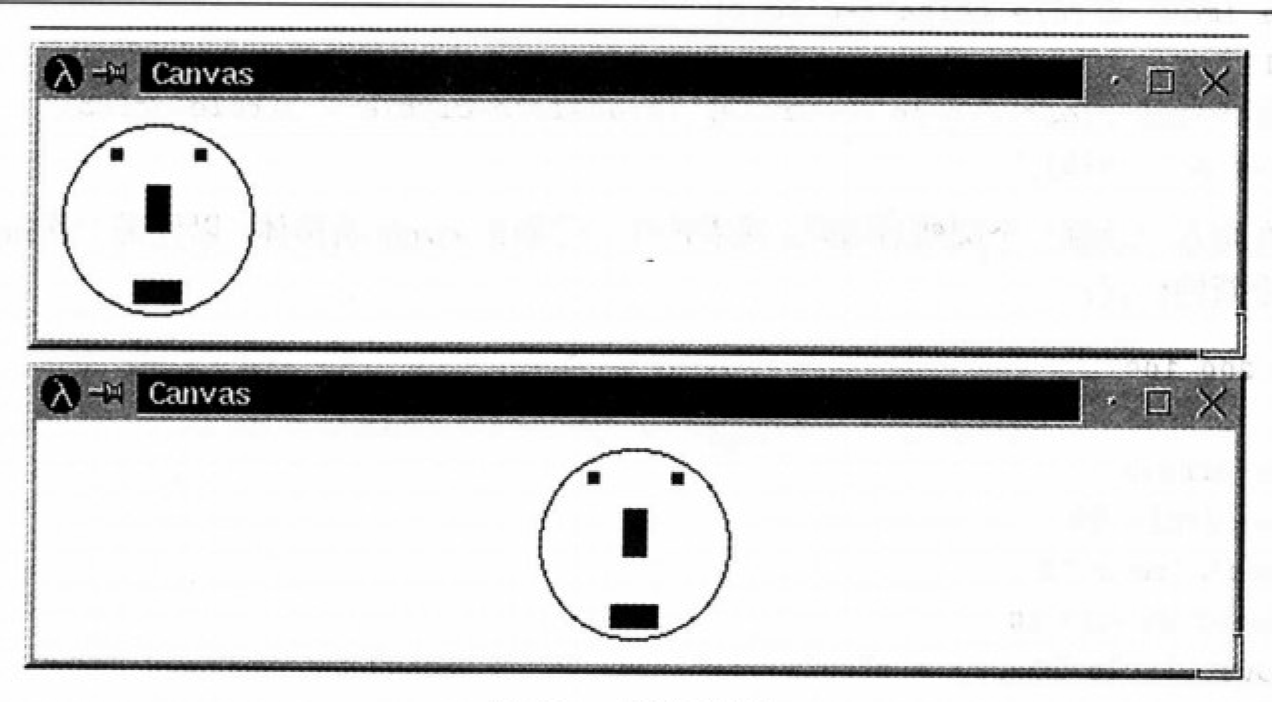


图 6.6　一张移动的脸

程序逐步求精是设计复杂程序的常用方法。当然，我们必须了解它的最终目标，并要牢记它，以便成功地使用它。因此，一个较好的方法是先写下一个计划表，每次精化之后再重新考虑它，我们将在第 16 章对其进行讨论。

习题

习题 6.6.1　给出一个带有颜色的圆 *circle* 的结构体定义和数据定义。一个圆包括三部分信息：圆心、半径和圆周的颜色。其中第一个信息是 *posn* 结构体，第二个信息是数，第三个信息是（颜色）符号。

开发模板 *fun-for-circle*,它是输入为 *circle* 的函数的框架，输出未定。

习题 6.6.2　使用模板 *fun-for-circle* 设计函数 *draw-a-circle*。该函数的输入为一个 *circle* 结构体，其效果是在屏幕上绘制一个相应的圆。在进行函数测试前，请使用(*start* 300 300)创建画布。

习题 6.6.3　使用模板 *fun-for-circle* 设计函数 *in-circle?*，该函数读入一个 *circle* 结构体和一个 *posn* 结构体，确定 *posn* 结构体表示的像素是否落在圆的内部。所有和圆心的距离小于或等于半径的像素都在圆内，其他的像素在圆外。

考虑图 6.7 中的圆，它的圆心为(make-posn 6 2)，半径为 1。标记为 A 的像素(make-posn 6 1.5)位于圆内；标记为 **B** 的像素(make-posn 8 6)位于圆外。

习题 6.6.4　使用模板 *fun-for-circle* 设计函数 *translate-circle*。该函数读入一个 *circle* 结构体和一个数 *delta*，输出另一个圆，其圆心位于原始圆的右边，与原始圆的圆心距离为 *delta*，该函数对画布没有影响。

几何平移：称一个几何体沿着直线移动为平移。

习题 6.6.5　使用模板 *fun-for-circle* 设计函数 *clear-a-circle*，该函数读入一个 *circle* 结构体，其效果是在画布上清除该圆。

习题 6.6.6　定义函数 *draw-and-clear-circle*，它按照 *circle* 结构体画一个圆，经过较短时间后，将其清除。教学软件包 draw.ss 提供有函数 *sleep-for-a-while*，它的输入为一个数，效果是计算机将休眠这些时间（单位为秒），该函数的返回值为 true。例如，执行(*sleep-for-a-while* 1)的效果是计算机休眠 1

秒钟。

下面的函数是在画布上每次一小步平移一个圆的关键部分：

```
;; move-circle : number circle -> circle
;; 绘制并消除一个圆，再平移 delta 个像素
(define (move-circle delta a-circle)
  (cond
    [(draw-and-clear-circle a-circle) (translate-circle a-circle delta)]
    [else a-circle]))
```

该函数在画布上绘制一个圆然后清除，接着产生一个新的 *circle* 结构体，以便另一次执行该函数时将它移到一个新的位置：

```
(start 200 100)

(draw-a-circle
  (move-circle 10
    (move-circle 10
      (move-circle 10
    (move-circle 10 ... a circle ...)))))
```

该表达式将一个圆移动了 4 次，每次 10 个像素，并在画布上显示这一移动。最后一个 *draw-a-circle* 是必需的，否则将无法看到最后的圆。

习题 6.6.7 给出有色长方形的结构体定义和数据定义。一个长方形包含四个特征信息：左上角位置、宽度、高度以及填充颜色。第一个信息是 *posn* 结构体，第二和第三个信息都是数值，第四个是颜色。

设计模板 *fun-for-rect*，用于描述输入为 *rectangle* 的函数，输出未定。

习题 6.6.8 使用模板 *fun-for-rect* 设计函数 *draw-a-rectangle*，该函数读入一个 *rectangle* 结构体，效果是在屏幕上绘制该长方形。与圆相反，长方形是实心的，以相应的颜色填充。记住在测试程序前使用(*start* 300 300)创建一个画布。

习题 6.6.9 使用模板 *fun-for-rect* 设计函数 *in-rectangle?*，该函数读入一个 *rectangle* 结构体和一个 *posn* 结构体，判断 *posn* 结构体表示的像素是否位于长方形内部。如果某个像素与长方形的左上角的坐标距离皆是正数，并且小于等于长方形的宽度和高度时，它位于长方形的内部，否则位于外部。

考虑图 6.7 中的长方形，它的主要参数为(make-posn 2 3)、3 和 2，点 *C* 位于长方形的内部，点 *B* 位于长方形的外部。

习题 6.6.10 使用模板 *fun-for-rect* 设计函数 *translate-rectangle*，该函数读入一个 *rectangle* 结构体和数 *delta*，返回另一个长方形，该长方形位于原长方形的右边，左上角和原长方形左上角的距离为 *delta*。该函数对画布没有影响。

习题 6.6.11 使用模板 *fun-for-rect* 设计函数 *clear-a-rectangle*，该函数读入一个 *rectangle* 结构体，执行该函数的效果是清除画布上相应的长方形。

习题 6.6.12 以下是 *move-rectangle* 函数：

```
;; move-rectangle : number rectangle -> rectangle
;; 绘制并清除一个长方形，然后平移 data 像素
(define (move-rectangle delta a-rectangle)
  (cond
    [(draw-and-clear-rectangle a-rectangle)
     (translate-rectangle a-rectangle delta)]
    [else a-rectangle]))
```

该函数先在画布上绘制一个长方形，然后清除，再平移 *delta* 个像素。

设计函数 *draw-and-clear-rectangle*，该函数绘制一个长方形，休眠一段时间，然后再清除。最后，创建一个长方形，并使用本习题中所定义的函数，将长方形移动 4 次。

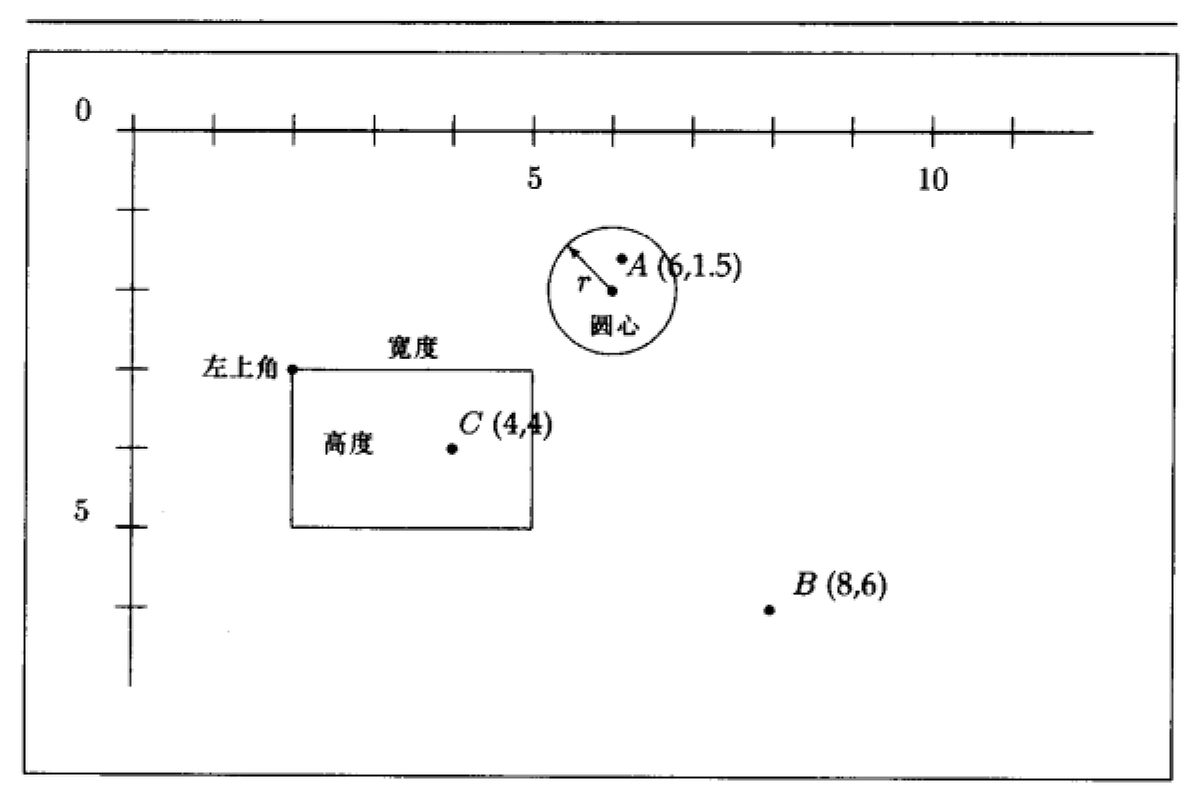


图 6.7　圆、长方形和像素

6.7　补充练习：刽子手游戏

刽子手游戏是一个两人参与的猜字游戏。其中一人先想象一个包含 3 个字母的单词[1]，并绘制绞刑架和套索，另一个人猜测第 1 个人所想象的单词，每次给出一个字符。如果猜错，第一个人则在图中添加人体的一部分：先是头、接着是身体、手臂和腿等。如果猜测和想象中的单词有一个或多个字母相同，该字母就在相应的位置上显示出来。当第二个人猜出整个单词或第一个人完成他的画时，游戏结束。

我们来设计一个程序，扮演游戏中的第一个人。该程序包括两个部分：一部分画图，另一部分确定第二个人是否猜对了单词中的字母，以及字母在单词中的位置。

习题

习题 6.7.1　设计函数 *draw-next-part*，该函数绘制人体的某一部分，函数的输入为下列七个符号之一：

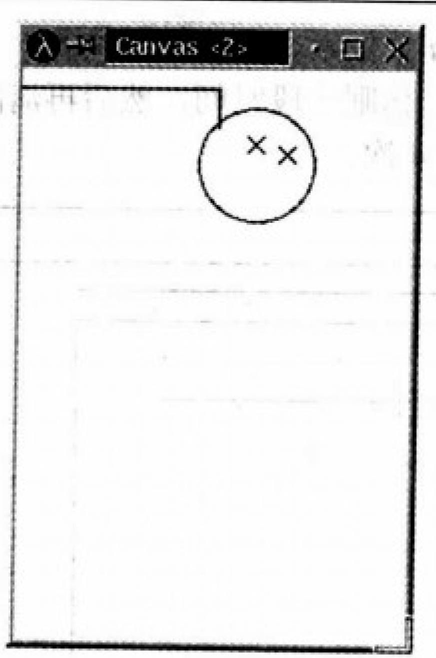
'right-leg　'left-leg　'left-arm　'right-arm　'body　'head　'noose

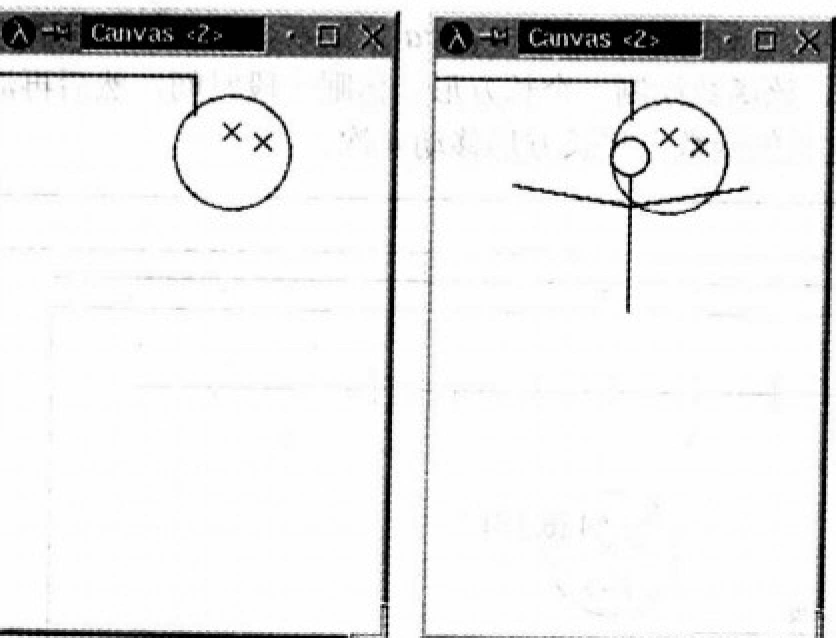
函数的返回值为 true，其效果是绘制相应的图形。图 6.8 是游戏中间过程的三个快照[2]。

提示：在 DrScheme 的 definitions 窗口的顶部增加(*start* 200 200)，然后从套索开始绘画，如果绘制人体的某一部分需要两个绘图操作，使用 and 表达式将它们组合起来。

[1] 在实际的游戏中，单词的长度是任意的。限制使用 3 个字符的目的是使游戏易于实现，我们将在练习 17.6.2 再次讨论这个游戏。

[2] 感谢 Johw Clements 先生提供 draw-next-part 的绘画版本。





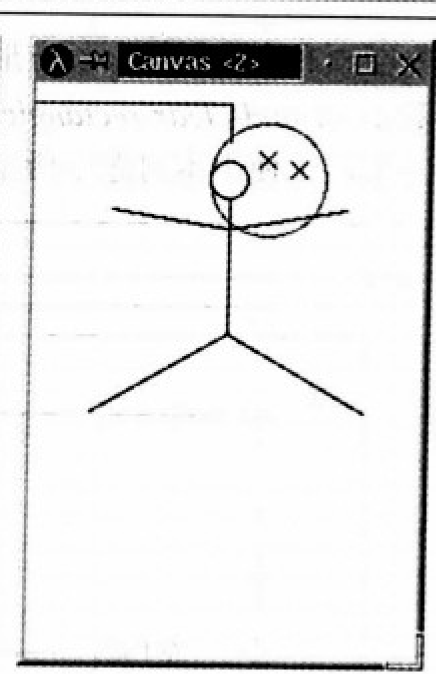


图 6.8 刽子手游戏的 8 个过程

第一个游戏者的第二个任务是确定另一个游戏参与者所猜测的字母是否出现在自己所想象的单词中，如果是的话，任务还包括显示该字符在哪里出现。设计诀窍要求在设计函数之前必须分析数据并提供数据定义。游戏的关键对象是单词和字母。一个单词包括 3 个字母。字母是从**'a** 到**'z** 的符号。然而，使用这些字母还不够，因为程序还必须维持一个单词，以记录第二个游戏参与者目前为止所猜中的字母。解决方案之一是使用一个特殊的“字母”，教学软件包中所使用的是'_。

习题

习题 6.7.2 给出表示由 3 个字母组成的单词的结构体定义和数据定义。

习题 6.7.3 设计函数 ***reveal***，该函数读入三个参数：

1. *chosen*，要猜测的单词；
2. *status*，状态单词，表示该单词的哪些部分已经被猜中；
3. 一个字符，当前的猜测。

函数返回一个新的状态单词，即包含通常字符和'_的单词。比较当前的猜测和被猜测单词的每个字符，从而求出新的状态单词的值：

1. 如果当前猜测等于被猜测单词的某个字符，用当前猜测代替新状态单词中的相应字符；
2. 否则，状态单词不变。

使用下列例子对函数进行测试：

```
(reveal (make-word 't 'e 'a) (make-word '_ 'e '_) 'u)
;; 预期值:
(make-word '_ 'e '_)

(reveal (make-word 'a 'l 'e) (make-word 'a '_ '_) 'e)
;; 预期值:
(make-word 'a '_ 'e)

(reveal (make-word 'a 'l 'l) (make-word '_ '_ '_) 'l)
;; 预期值:
(make-word '_ 'l 'l)
```

第一个例子是当前猜测与被猜单词不符，第二个例子是当前猜测在被猜单词出现，最后一个例子是当前猜测在被猜单词出现两次。

提示：(1)当一个定义变得较大而难以管理的时候，使用辅助函数；(2)函数 *reveal* 读入两个结构体和一个字符，这提示我们使用复合数据的设计诀窍。对于模板，最好是将选择表达式写成两列，每列对应于一个结构体。

当正确测试完函数 *draw-next-part* 和 *reveal* 之后，在 DrScheme 中将教学软件包设置为 hangman.ss，并计算

```
(hangman make-word reveal draw-next-part)
```

其中函数 *hangman* 随机选择一个包含 3 个字母的单词，弹出一个字符菜单，用户可以选择一个字符，按下 Check 按钮看是否猜对。将练习 6.7.1 中的测试注释掉，使绘制图形的效果不影响这里 *hangman* 函数的运行。



第7章 数据的多样性

前一章的讨论拓广了我们的数据世界。本章要处理一个包含布尔值、符号以及各种类型的结构体的世界，首先我们要给这样的一个世界制定一些规则。

到现在为止，函数只能处理下述四种数据[1]：

数： 数值信息；
布尔值： 真和假；
符号： 符号信息；
结构体： 复合信息。

有时候，函数必须处理这样一种数据类型，它既包含数，又包含结构体，甚至包含若干种不同类型的结构体。在这一章，我们将学习如何设计这种函数。另外，我们还将学习如何避免函数被不正确使用，如某些用户可能偶然地把某个绘制圆的函数作用于一个矩形。虽然这种用法与数据定义相悖，但是我们尚不知道如何在需要的时候保护我们的函数，预防不正确的用法。

7.1 数据混合与区分

前一章使用了包含两个分量的 *posn* 结构体来表示像素。如果有许多像素都位于 x 轴，我们可以简单地使用普通数值来表示它们，而使用 *posn* 结构体表示其他像素。

在图 7.1 所示的五个点中，有三个点位于 x 轴上，分别是 *C*、*D*、E，因此只有 *A* 和 *B* 两个点，需要使用两个坐标表示。使用新的点表示法可以更简洁地描述这些点：(make-posn 6 6)代表 *A*；(make-posn 1 2)代表 *B*；而 1、2、3 分别代表 *C*、*D*、*E*。

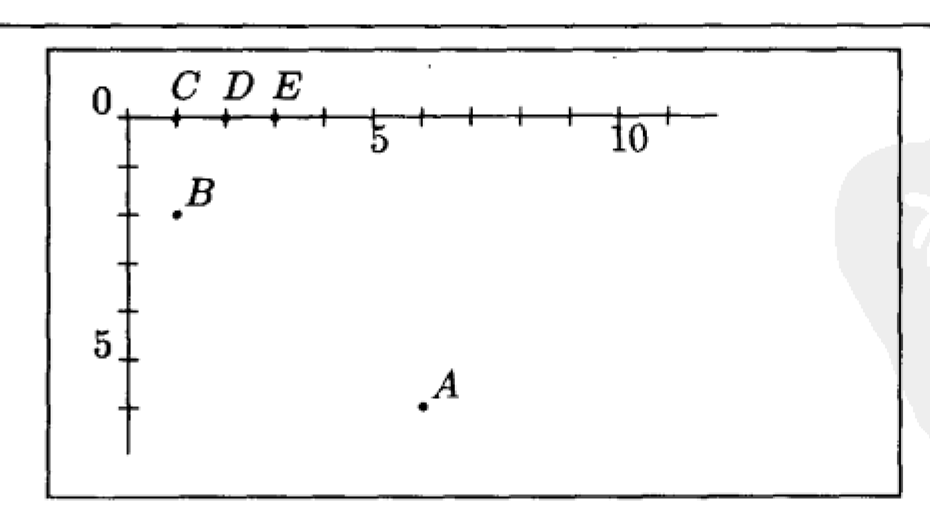


图 7.1　一个点集

现在，如果要定义函数 *distance-to-0*，该函数读入一个点，返回该点到原点的距离。这时会遇到一

[1] 我们还讨论过图像和字符串，但是这里我们忽略它们。

个问题：该函数可能会被应用于一个数，或者是一个 *posn* 结构体。根据输入数据类型的不同，*distance-to-0* 必须采用不同的方法计算该点到原点的距离。因此，需要使用一个 **cond** 表达式来区分这两种类型。不幸的是，现在还没有任何操作能够给出适当的条件。

为了解决这些问题，Scheme 提供了辨别数据形式的谓词（predicate），包括：

number?，读入一个任意值，如果是数，返回 **true**，否则返回 **false**；
boolean?，读入一个任意值，如果是布尔值，返回 **true**，否则返回 **false**；
symbol?，读入一个任意值，如果是符号，返回 **true**，否则返回 **false**；
struct?，读入一个任意值，如果是结构体，返回 **true**，否则返回 **false**。

对于每种结构体定义，Scheme 都将引入一个谓词。假设 Definitions 窗口中包含了如下的结构体定义[1]：

```
(define-struct posn (x y))

(define-struct star (last first dob ssn))

(define-struct airplane (kind max-speed max-load price))
```

那么，Scheme 将引入如下三个谓词：

posn?，读入一个任意值，如果该值是一个 *posn* 结构体，返回 **true**，否则返回 **false**；
star?，读入一个任意值，如果该值是一个 *star* 结构体，返回 **true**，否则返回 **false**；
airplane?，读入一个任意值，如果该值是一个 *airplane* 结构体，返回 **true**，否则返回 **false**。

借助它们，函数可以区分结构体和数，也可以区分 *posn* 结构体和 *airplane* 结构体。

习题

习题 7.1.1　手工计算下列表达式：

```
(number? (make-posn 2 3))
(number? (+ 12 10))
(posn? 23)
(posn? (make-posn 23 3))
(star? (make-posn 23 3))
```

使用 DrScheme 检查你的答案。

现在可以开发函数 *distance-to-0* 了，让我们从数据定义开始：

> *pixel-2* 是下列二者之一：
> 1. 数。
> 2. *posn* 结构体。

以下是合约、用途说明和函数头部：

```
;; distance-to-0 : pixel-2 -> number
;; 计算 a-pixel 到原点的距离
(define (distance-to-0 a-pixel) ...)
```

[1] posn 结构体由 DrScheme 的教学语言自动提供，用户不应该再对它进行定义。

正如前文所述，该函数区分两种输入类型，这可由一个 cond 表达式实现：

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) ...]
    [(posn? a-pixel) ...]))
```

cond 的两个条件分别对应函数的两种可能输入。如果第一个条件成立，输入就是位于 *x* 轴的像素。否则，该像素就是 *posn* 结构体。对于第二个 cond 子句，我们还知道输入包含了两个元素：*x* 坐标和 *y* 坐标。为了提醒自己，我们在模板中添上两个选择器表达式：

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) ...]
    [(posn? a-pixel) ... (posn-x a-pixel) ... (posn-y a-pixel) ... ]))
```

现在要完成这个函数就容易了。如果输入是个数，那么它就是到原点的距离；如果输入是个结构体，则必须使用原来的公式求出该点到原点的距离：

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) a-pixel]
    [(posn? a-pixel) (sqrt
                 (+ (sqr (posn-x a-pixel))
                 (sqr (posn-y a-pixel))))]))
```

再来考虑第二个例子。假设要编写一些处理几何图形的函数。其中一个函数计算某个图形的面积，另一个函数计算边长，第三个函数绘制该图形。为了简单起见，假设图形只包含正方形和圆形，交且都由位置（一个 *posn* 结构体）和大小（一个数）表示。

两种图形的信息都必须用结构体表示，因为二者都有多个属性。下面是它们的结构体定义：

```
(define-struct square (nw length))
(define-struct circle (center radius))
```

数据定义如下：

> *Shape*（图形）是下列二者之一：
> 1.*cirde* 结构体：
> **(make-circle** *p s***)**
> 其中 *p* 是 *posn* 结构体，*s* 是数；
> 2.*square* 结构体：
> **(make-square** *p s***)**
> 其中 *p* 是 *posn* 结构体，*s* 是数。

设计诀窍的下一步是构造例子，从输入例子开始：

1. (make-square (make-posn 20 20) 3)；
2. (make-square (make-posn 2 20) 3)；
3. (make-circle (make-posn 10 99) 1)。

为了构造出输入与输出之间关系的例子，我们需要知道函数的用途。如果函数 *perimeter* 计算图形的周长。根据几何学知识，正方形的周长是其边长的四倍，圆形的周长是直径乘以 π，而直径是半径的两

倍[1]。因此，上述三个例子的周长分别是：12、12 及 6.28（近似值）。

按照设计诀窍和前述的 *distance-to-0*，我们从如下框架开始设计函数：

```
;; perimeter :  shape -> number
;; 计算图形 a-shape 的周长
(define (perimeter a-shape)
  (cond
    [(square? a-shape) ... ]
    [(circle? a-shape) ... ]))
```

可以看出函数先判定 *a-shape* 属于哪种类型。

另外，由于两种可能的输入都是结构体，我们还可以在每个 cond 子句中添加两个选择器表达式：

```
;; perimeter :  shape -> number
;; 计算 a-shape 的周长
(define (perimeter a-shape)
  (cond
    [(square? a-shape)
     ... (square-nw a-shape) ... (square-length a-shape) ...]
    [(circle? a-shape)
     ... (circle-center a-shape) ... (circle-radius a-shape) ...]))
```

选择器表达式提示函数可用的数据。

现在，把数学公式转换成 Scheme 表达式，填充两个子句后变为：

```
(define (perimeter a-shape)
  (cond
    [(square? a-shape) (* (square-length a-shape) 4)]
    [(circle? a-shape) (* (* 2 (circle-radius a-shape)) pi)]))
```

由于图形的位置不影响它的周长，所以删去了模板中与 *nw* 和 *center* 相关的选择器表达式。

习题

习题 7.1.2　用例子对函数 *perimeter* 进行测试。

习题 7.1.3　开发函数 *area*，该函数读入一个圆形或者正方形，计算它的面积。考虑能不能通过将名字改为 *area* 而使用 *perimeter* 的模板？

7.2　设计处理混合数据的函数

上一节函数设计的过程表明，设计诀窍还需要进一步修正。具体来说，数据分析、模板以及主体的定义需要修正。

数据分析和设计：分析问题表述的任务之一就是判断该问题有没有涉及不同类型的数据，这类数据通常称为混合数据（Mixed Data），或称为数据的联合体（Union）。换句话说，数据分析必须考虑多个因素。第一，必须确定问题提及多少种不同的数据类型，它们各自的属性又是什么。如果存在多个不同的数据类型，就将它们组成混合数据；第二，必须理解涉及的对象有没有多个属性，如果某个对象有多

[1] 圆的周长也称为圆周。

个属性，就用结构体来表示它。因此，数据定义可能包含多个子句，列举多种可能的情况。事实上，数据分析可能会生成一个多层的数据定义。

上一节中的例子处理了两种不同类型的图形，而每种图形又有多个属性，可以使用如下数据定义来描述这种思想：

> *shape*（图形）是下列二者之一：
> 1.*arde* 结构体：
> **(make-circle** *p s*)
> 其中 *p* 是 *posn* 结构体，*s* 是数；
> 2.*square* 结构体：
> **(make-square** *p s*)
> 其中 *p* 是 *posn* 结构体，*s* 是数。

这表明 *shape* 是上述两种数据子类型之一。

为了使数据定义有意义，必须给出区分不同子类型的条件。也就是说，如果 *x* 是所定义的类型中的一个数据，那么必须能够使用内置谓词或者用户自定义谓词来区分它是属于哪个子类型。在这个例子中，所需的条件是(square? *x*)和(circle? *x*)。

模板：模板就是把输入数据转换成 Scheme 表达式。例如，一个数据定义列举了多个不同的事物。第一步是写下一个 cond 表达式，其子句数量与数据定义所包含的不同类型的数据种类的数目相等；第二步是给每一个子句加上条件，与数据定义中相应的子类型相对应，当输入属于该子类型时，条件成立。

以下是该例子的模板：

```
;; f : shape -> ???
(define (f a-shape)
  (cond
    [(square? a-shape) ...]
    [(circle? a-shape) ...]))
```

模板省略了输出和用途说明，因此模板与函数输出及用途说明之间没有任何的联系。

一旦描述了带有条件的模板，就可以一个 cond 子句一个 cond 子句地修正模板。如果一个子句的用途是处理原子信息，那么已经完成了修正；如果一个子句的用途是处理复合数据，那么在模板中添上适当的选择器表达式。

再一次用例子阐明这种思想：

```
(define (f a-shape)
  (cond
    [(square? a-shape)
     ... (square-nw a-shape) ... (square-length a-shape) ...]
    [(circle? a-shape)
     ... (circle-center a-shape) ... (circle-radius a-shape) ...]))
```

主体：模板将任务分割成多个子任务。现在可以单独处理每一个 cond 子句了。事实上，只需简单地考虑如果输入是某种类型的数据，输出应该是什么就可以了。因而在处理某个特定的子句时，可以忽略其他情况。

假设要定义一个计算图形周长的函数。从填写模板中的空缺开始：

```
;; perimeter : shape -> number
;; 计算 a-shape 的周长
(define (perimeter a-shape)
```

```
(cond
  [(square? a-shape) (* (square-length a-shape) 4)]
  [(circle? a-shape) (* (* 2 (circle-radius a-shape)) pi)]))
```

图 7.2 是函数开发过程的概括。

```
;; 数据定义:
(define-struct circle (center radius))
(define-struct square (nw length))
;; shape 是下列二者之一
;; 1. 结构体:  (make-circle p s)
;;     其中 p 是 posn 结构体，s 是数。
;; 2. 结构体:  (make-square p s)
;;     其中 p 是 posn 结构体，s 是数。

;; 合约、用途说明、函数头部:
;; perimeter : shape -> number
;; 计算 a-shape 的周长

;; 例子:  参见测试

;; 模板:
;; (define (f a-shape)
;;   (cond
;;     [(square? a-shape)
;;   ... (square-nw a-shape) ... (square-length a-shape) ...]
;;     [(circle? a-shape)
;;     ... (circle-center a-shape) ... (circle-radius a-shape) ...]))

;; 定义:
(define (perimeter a-shape)
  (cond
    [(circle? a-shape)
     (* (* 2 (circle-radius a-shape)) pi)]
    [(square? a-shape)
     (* (square-length a-shape) 4)]))

;; 测试:  (即例子)
(= (perimeter (make-square ... 3)) 12)
(= (perimeter (make-circle ... 1)) (* 2 pi))
```

图 7.2　处理混合数据的函数设计：一个完整的例子

图 7.3 给出了处理混合数据函数的设计总结。

比较一下第 2.5 节、第 4.4 节、第 6.5 节和本节中的设计诀窍，我们发现数据分析和模板设计变得越来越重要了。不理解函数读入的数据类型，就无法正确设计函数。反之，如果理解了数据定义，正确建立了模板，那么要修改或者扩展函数就容易了。例如，如果要给 *circle* 的表示法添加新的信息，那么只有与圆形相关的那些 cond 子句需要修改。类似地，如果我们想在数据定义中添加了一种新的图形，比如说矩形，则只需在主体中添加一条新的 cond 子句即可。

阶段	目标	任务
数据分析和设计	数据定义	确定问题数据中有多少种不同的对象 • 在数据定义中枚举这些对象 • 如果它是一种复合数据，对每个成分阐明它的数据定义
合约、用途说明和函数头部	给函数命名； 说明输入和输出数据的类型； 描述函数的用途说明； 阐明函数头部	给函数命名，说明输入数据的类型、输出数据的类型，指出函数的目的； ;;*name: in1in2… ->out* ;;从 x1…计算… （define(namex1 x2…)…）
例子	使用例子刻划输入和输出间的关系	创建说明输入输出关系的例子 • 确保每一个子类有一个例子
模板	阐明程序框架	对于每个子类引入一个 **cond** 表达式 • 使用内置谓词、预定义谓词为每种情况阐明一个条件
主体	定义函数	假定条件为真，给每个 **cond** 行设计一个 Scheme 表达式
测试	发现错误（拼写错误和逻辑错误）	将函数应用于例子中的输入 • 检查程序输出是否与预期的值相符

图 7.3 处理混合数据的函数（图 2.2、图 6.5 的精化）

习题

习题 7.2.1 给出动物园中动物的结构体和数据定义，涉及的动物包括：

蜘蛛，属性包括所剩的腿的数目（假设蜘蛛可能会在意外事故中失去一些腿）和运输它们时所需的空间大小；

大象，属性只包括其在运输时所需的空间大小；

猴子，属性包括智力和其在运输时所需的空间大小。

再开发一个读入动物的函数模板。

开发函数 *fits?*，该函数读入一个动物和一个笼子的容积，判断笼子能否容下动物。

习题 7.2.2 城市交通管理处负责管理各种交通工具。给出交通工具的结构体和数据定义，至少包含公共汽车、豪华轿车、客车及地铁。并给每种交通工具附上至少两种属性。

开发一个读入交通工具的函数模板。

7.3 再论函数复合

在分析问题时，可能会希望逐步设计数据表示，当涉及多个不同的对象时，这一点尤为突出。与设计一个大型的数据定义相比，先设计多个小型数据定义，再把它们组合起来要容易得多。

回过头来看图形的例子。设计单独的一个数据定义，也可以从两个数据定义出发，每个数据定义分别表示一个图形：

> *circle*（圆形）是结构体：
>
> **(make-circle** *p s***)**
>
> 其中 *p* 是 *posn* 结构体，*s* 是数。

> *square*（正方形）是结构体：
>
> **(make-square *p s*)**
>
> 其中 *p* 是 *posn* 结构体，*s* 是数。

一旦有了基本的数据定义，并且通过例子或通过编写简单的函数理解了它们，就可以将它们组合起来。例如，使用上述两个数据定义，我们可以引入如下数据定义：

> *shape*（图形）是下列二者之一：
>
> 1. *Circle*。
> 2. *Square*。

假设需要设计一个读入 *shape* 的函数。第一步，构造一个 cond 表达式，其中的每个条件对应数据定义中的一个部分：

```
;; f : shape -> ???
(define (f a-shape)
  (cond
    [(circle? a-shape) ...]
    [(square? a-shape) ...]))
```

这个数据定义引用了另外两条数据定义，根据第 3.1 节中关于函数复合的原则，第二步自然是把参数传递给辅助函数：

```
(define (f a-shape)
  (cond
    [(circle? a-shape) (f-for-circle a-shape)]
    [(square? a-shape) (f-for-square a-shape)]))
```

这就需要我们设计两个辅助函数，*f-for-circle* 和 *f-for-square*，当然还有它们的模板。

如果遵循此建议，我们会得到一组共三个函数，每个函数对应一个数据定义。图 7.4 的右侧一栏列出了这样设计的程序。作为对比，左侧的一栏给出了原来的程序。在这两种情况下，所得到的函数数目都与数据定义一样多。另外，右侧一栏中函数间的引用也与对应的数据定义间的引用相对应。虽然现在这种一一对应看似平凡，但是，当我们学习到更复杂的数据定义方法时，它就显得非常有用了。

习题

习题 7.3.1 分别修改两个版本的 perimeter 函数，使它们能够处理矩形。根据需要，一个矩形的描述包括它的左上角、长度和宽度。

```
;; 数据定义：
(define-struct circle (center radius))
(define-struct square (nw length))
;; shape 是下列二者之一
;; 1. 结构体：(make-circle p s)
;;    其中 p 是 posn 结构体，s 是数；
;; 2. 结构体：(make-square p s)
;;    其中 p 是 posn 结构体，s 是数。
```

```
;; 数据定义：
(((define-struct circle (center radius))
;; circle 是结构体：
;;    (make-circle p s)
;; 其中 p 是 posn 结构体，s 是数；

(define-struct square(nw length))
;; square 是结构体：
;;    (make-square p s)
;; 其中 p 是 posn 结构体，s 是数。

;; shape 是下列二者之一
;; 1. circle，
;; 2. square。
```

```
;; 最终的定义：
;; perimeter  :  shape -> number
;; 计算 a-shape 的周长
(define (perimeter a-shape)
  (cond
    [(circle? a-shape)
     (* (* 2 (circle-radius a-shape)) pi)]
    [(square? a-shape)
     (* (square-length a-shape) 4)]))
```

```
;; 最终的定义：
;; perimeter  :  shape -> number
;; 计算 a-shape 的周长
(define (perimeter a-shape)
  (cond
    [(circle? a-shape)
     (perimeter-circle a-shape)]
    [(square? a-shape)
     (perimeter-square a-shape)]))

;; perimeter-circle  :  circle -> number
;; 计算 a-circle 的周长
(define (perimeter-circle a-circle)
  (* (* 2 (circle-length a-circle)) pi))

;; perimeter-square  :  square > number
;; 计算 a-square 的周长
(define (perimeter-square a-square)
  (* (square-length a-square) 4))
```

图 7.4　定义 perimeter 的两种方法

7.4　补充练习：图形的移动

第 6.6 节开发了绘制、平移以及删除圆形和矩形的函数。如同前一节所看到的那样，应该把这两种数据类型看作为是图形的子类型，这样只需绘制、平移及删除图形就可以了。

习题

习题 7.4.1　给出 *shape* 类型的一般数据定义，该类型至少包含第 6.6 节中的圆形和矩形。开发输入为 *shape* 类型的函数模板 *fun-for-shape*。

习题 7.4.2　使用模板 *fun-for-shape* 设计 *draw-shape*，该函数读入一个 *shape* 结构体，再把它绘制到画布上。

习题 7.4.3　使用模板 *fun-for-shape* 设计 *translate-shape*，该函数读入一个 *shape* 结构体和一个数 *delta*，生成一个图形，其关键位置在 x 方向上平移了 *delta* 个像素。

习题 7.4.4　使用模板 *fun-for-shape* 设计 *clear-shape*，该函数读入一个 *shape* 结构体，将其从画布上删除，返回 **true**。

习题 7.4.5　设计函数 *draw-and-clear-shape*，该函数读入一个 *shape* 结构体，先绘制相应的图形，

等待一段时间，再删除它。如果所有的效果都得以完成，函数返回 true。

习题 7.4.6　设计函数 *move-shape*，该函数在画布上移动一个图形。函数的输入为一个数（*delta*）和一个图形。函数先绘制然后删除图形，并返回一个新的、平移了 *delta* 像素的图形。多次使用这个函数，就可以在画布上平移一个图形。

7.5 输 入 错 误

考虑下面的函数：

```
;; area-of-disk : number -> number
;; 计算半径为 r 的圆盘的面积
(define (area-of-disk r)
  (* 3.14 (* r r)))
```

如果有人要使用这个函数来完成他的几何作业，但他在使用这个函数的时候，意外地将其应用于一个符号，而不是数。发生这种情况时，函数会停止运行，并给出一条古怪的的出错消息：

```
> (area-of-disk 'my-disk)
*: expects type <number> as 1st argument, given: 'my-disk; ...
```

使用谓词就可以避免此类问题。

如果要把函数提供给他人使用，为了防止这种意外，应当定义自带检查的函数。一般来说，检查函的输入是任意的 Scheme 值：数、布尔值、符号或者是结构体。对于所有在原先函数中有定义的类型的值，检查函数就把值传递给原先函数；对于其他值，检查函数将给出错误消息。具体来说，*checked-area-of-disk* 读入任意一个 Scheme 值，如果它是数，就使用 *area-of-disk* 来计算圆盘的面积，否则，就停止运行，并产生一个错误消息。

枚举所有 Scheme 值的类型，检查函数的模板应该如下：

```
;; f : Scheme-value -> ???
(define (f v)
  (cond
    [(number? v) ...]
    [(boolean? v) ...]
    [(symbol? v) ...]
    [(struct? v) ...]))
```

每一个子句对应于一种可能的输入类型。如果要区分不同的结构体，可以适当地扩展最后一个子句。

就 *area-of-disk* 而言，能使用的只是第一个子句，对于其他情况，必须产生一个错误消息。Scheme 使用 error 来产生错误消息。error 读入一个符号和一个字符串。下面是一个例子：

```
(error 'checked-area-of-disk "number expected")
```

checked-area-of-disk 的完整定义是：

```
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [(boolean? v) (error 'checked-area-of-disk "number expected")]
    [(symbol? v) (error 'checked-area-of-disk "number expected")]
```

```
      [(struct? v) (error 'checked-area-of-disk "number expected")]))
```

使用 **else** 还可以大大简化该函数：

```
;; checked-area-of-disk : Scheme-value -> number
;; 如果 v 是数的话，计算半径为 v 的圆盘的面积
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error 'checked-area-of-disk "number expected")]))
```

当然，这种简化并不总是可行的，有时需要重排 cond 子句的顺序。

若要把程序分发给其他人，编写检查函数并简化是相当重要的。不过，设计能够正常工作的程序更为重要。因此，本书将集中精力于正确程序的设计过程，而并不强调检查函数的编写。

习题

习题 7.5.1 *area-of-disk* 的自检查版本还可以要求函数的参数是一个正数，而不仅仅只是一个数。按照这个要求修改 *checked-area-of-disk*。

习题 7.5.2 设计函数 *profit*（图 3.1）、*is-between-5-6?*（第 4.2 节）、*reply*（第 5 章）、*distance-to-0*（第 6.1 节）和 *perimeter*（图 7.4 的左栏）的自检查版本。

习题 7.5.3 观察以下结构体和数据定义：

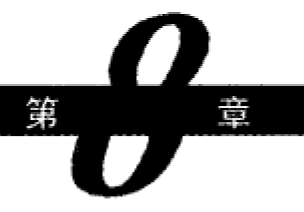
```
(define-struct vec (x y))
```

> *vec* 是结构体：
> **(make-vec** *x y***)**
> 其中 *x* 和 *y* 都是正数。

设计函数 *checked-make-vec*，其可以被理解成基本操作 *make-vec* 的自检查版本，该函数确保 *make-vec* 的参数都是正数，而不是任意的数，也可以认为 *checked-make-vec* 是非正式的数据定义。



语法和语义



到现在为止，我们已经初步了解了 Scheme 语言。就像蹒跚学步的幼童，我们学习了 Scheme 语言的词汇，理解了它的直观含义，还学会了一些遣词造句的基本规则。不过，要真正有效地使用一种语言来表达思想（无论是像英语这样的自然语言，还是像 Scheme 这样的人工语言）都需要对语言的词汇、语法和语义进行学习。

在许多方面，程序设计语言很像自然语言，它也有词汇和语法。在 DrScheme 中，词汇就是那些“基本单词”，也就是我们用来“遣词语句”的对象。在程序设计语言中，语句是表达式，或者是函数；文法描述了如何用单词来组成语句。在程序设计领域，我们使用术语“语法”来表示程序设计语言中的词汇和文法。

无论是在自然语言中还是在程序设计语言中，并非所有符合文法的语句都是有意义的。例如，在自然语言中，“猫是滚圆的”是一句有意义的句子，但句子“砖头是汽车”，虽然完全符合文法，却没有意义。要判断一句语句有没有意义，必须研究语句和单词的含义，即语义。自然语言通常借用更基本的词汇和句子来解释某个单词的意思；对于外来语言，一般使用更简单的词汇来解释一个单词，或者干脆把这个单词翻译成我们熟悉的母语词汇；对于程序设计语言，说明某一语句的含义也有好几种方法。本书先对大家所熟悉的算术和代数规则进行扩展，然后使用它来讨论 Scheme 程序的含义。毕竟，计算就是从算术运算开始的，况且我们也应该理解数学和计算之间的关系。

本章前三节介绍一个 Scheme 子集的词汇、语法和含义，这一子集虽小，但表示能力相当强大。在对 Scheme 程序的含义有了新的认识之后，第四节将继续讨论程序运行错误。本章后三节对 **and** 表达式、**or** 表达式、变量定义和结构体再次进行了讨论。

8.1 Scheme 的词汇

基本的 Scheme 词汇有五类。计算机科学家一般使用如图 8.1 所示的形式来表示词汇，图中竖线（“|”）对基本词例进行了分隔，而省略号则表明，在一个种类中还有同类的东西。

<var>	=	*x* | *area-of-disk* | *perimeter* | ...
<con>	=	**true** | **false**
		'a | **'doll** | **'sum** | ...
		1 | **-1** | **3/5** | **1.22** | ...
<prm>	=	+ | - | ...

图 8.1　Beginning Student Scheme：核心词汇

第一种类别是变量，也就是函数和值的名字；第二种类别是常量，包括布尔值、符号和数值常量。正如前文所提到的，Scheme 有一个强大的数值系统，要介绍这个系统，最好的方法是使用例子逐步地进

行。最后一个类别是基本操作，也就是 Scheme 提供的基本函数。虽然目前不能完整地给出这个类别，不过可以随着它们的出现而逐一介绍。

要给 Scheme 语句分类，需要三个关键字：define、cond 和 else。这些关键字本身并没有意义，它们就像自然语言中的标点符号，其目的是帮助程序员区分不同的语句。要注意的是，关键字不可以用作变量名。

8.2 Scheme 的文法

与许多其他程序设计语言不同，Scheme 的文法很简单。图 8.2 给出了完整的 Scheme 文法[1]。文法定义了两种类型的语句：*<def>*和*<exp>*，即定义和表达式。文法并没有说明语句中各项之间如何隔开，不过按照惯例，每项之后至少放上一个空格，除非该项后紧跟的是右扩号“)”。对于分隔符的使用，Scheme 比较灵活，除了可以使用一个空格外，也可以使用多个空格、换行符或者分页符。

<def>	=	(**define** (*<var>* *<var>* ...*<var>*) *<exp>*)
<exp>	=	*<var>*
		| *<con>*
		| (*<prm>* *<exp>* ...*<exp>*)
		| (*<var>* *<exp>* ...*<exp>*)
		| (**cond** (*<exp>* *<exp>*) ...(*<exp>* *<exp>*))
		| (**cond** (*<exp>* *<exp>*) ...(*else* *<exp>*))

图 8.2 Beginning Student Scheme 文法

上述两条文法描述了简单句和复合句的结构。所谓复合句，就是由多个语句组成的语句。例如，一个函数定义由“(”、关键字 define、另一个“(”、一个非空的变量序列、一个“)”、一个表达式以及与第一个左括号对应的“)”组成。关键字 define 把定义和表达式区分开。

表达式的类别有六种：变量、常量、基本操作、函数调用以及两种 cond 表达式，其中后四种表达式由其他表达式组成。关键字 cond 使条件表达式和基本操作应用、函数调用相互区分。

这是表达式的三个例子：'all、*x* 和(*x* *x*)。第一个表达式属于符号类型；第二个表达式是一个变量，并且每个变量都是表达式；因为 *x* 是变量，因此第三个表达式是函数调用。

相反，下列带括号的语句不是表达式：(*f* define)、(cond *x*)和()。第一个语句部分符合函数调用的外形，但是它把 **define** 当作变量来使用。第二个语句并不是一个正确的 cond 表达式，因为第二项只包含了一个变量，而不是包含在括号内的一对表达式。最后一个语句只是一对括号，但文法要求每一个左括号后都要紧跟一个不是右括号的对象。

习题

习题 8.2.1 为什么以下语句

1．*x*　　　2．(= *y* *z*)　　　3．(= (= *y* *z*) 0)

是符合文法的表达式？并解释为什么下列语句不是合法的表达式：

1．(3 + 4)　　　2．empty?(*l*)　　　3．(*x*)

[1] 这个文法只描述了我们目前已看到的部分 Scheme（不包括变量和结构体定义），不过它已经覆盖了整个语言相当大的一个子集。Scheme 比这个文法要大一些，本书的后续章节会继续介绍 Scheme 的其他部分。

习题 8.2.2 为什么以下语句是符合文法的定义？

1. `(define (f 'x) x)` 2. `(define (f x) y)` 3. `(define (f x y) 3)`

并解释为什么下列语句不是合法的定义：

1. `(define (f 'x) x)` 2. `(define (f x y z) (x))` 3. `(define (f) 10)`

习题 8.2.3 判断下列语句是否合法：

1. `(x)` 2. `(+ 1 (not x))` 3. `(+ 1 2 3)`

并解释它们为什么合法，或者为什么不合法。

习题 8.2.4 判断下列语句是否合法：

1. `(define (f x) 'x)` 2. `(define (f 'x) x)` 3. `(define (f x y) (+ 'y (not x)))`

并解释它们为什么合法，或者为什么不合法。

文法术语：复合句的成分各有称谓。为了方便，这里介绍一些有用的名称。函数定义中的第二个部分，也就是一个非空的变量序列，被称为函数的头部。相应地，定义中的表达式部分被称为主体。在头部，在第一个变量之后的所有变量被称为函数的参数。

(**define** (*<function-name>* *<parameter>* ...*<parameter>*) *<body>*)

(*<function>* *<argument>* ...*<argument>*)

(**cond** (*<question>* *<answer>*) **<cond-*clause*>** ...)

图 8.3 语法命名惯例

有人把定义看作为数学函数的定义，使用术语“左部”来表示定义的头部，“右部”表示主体。根据同样的理由，函数调用的第一个成分被称为函数，其余部分被称为参数。

最后，cond 表达式由 cond 子句组成的。每个子句包含两个表达式：问题（question）和答案（answer）。问题也叫做条件(cond-tion)

图 8.3 是这些惯例的总结。

8.3 Scheme 的含义

一个合法的 DrScheme 程序包含了两个部分：函数定义序列（位于定义窗口之中）和交互序列（位于交互窗口之中）。交互就是需要计算的 Scheme 表达式，一般涉及在定义窗口定义的函数。

在计算表达式的时候，DrScheme 惟一所做的事就是使用算术和代数规则，把表达式转化为值。在普通数学课程中，值就是数。在这里，我们认为符号、布尔值以及所有的常量都是值，即

<val> = *<con>*

因此值是表达式的一个子集。

定义了值的集合，要说明计算规则就容易了。计算规则的来源有二：一是算术知识，另一是代数。首先，需要算术规则来说明基本操作，这类规则有无数多种，如：

```
(+ 1 1) = 2
(- 2 1) = 1
```

但是，Scheme 的“算术”并不仅仅处理数，还处理布尔值、符号和表，所以还包含如下规则：

```
(not true) = false
(symbol=? 'a 'b) = false
(symbol=? 'a 'a) = true
```

其次，需要一条代数规则，用来说明用户自己定义的函数的计算过程。假设在 Definitions 窗口中包含了以下定义：

```
(define (f x-1 ... x-n)
  exp)
```

其中 *f*，*x-1*，……，*x-n* 是变量，exp 是某个（合法的）表达式。那么函数调用的计算规则就是：

```
(f v-1 ... v-n)         = Exp，其中所有的 x-1 ……  x-n 都被替换成 v-1 …… v-n
```

这里的 *v-1* …… *v-n* 是一个和 *x-1* …… *x-n* 一样长的序列。

这是一条概括性的规则，所以，最好观察一下具体的例子。比方说，定义是：

```
(define (poly x y)
  (+ (expt 2 x) y))
```

那么调用(*poly* 3 5)的计算过程为：

```
  (poly 3 5)
= (+ (expt 2 3) 5))
  ;; 这一行就是(+ (expt 2 x) y)，
  ;; 其中的 x 被替换成 3，y 被替换成 5。
= (+ 8 5)
= 13
```

其中最后两步是普通的算术计算。

最后，还需要计算 cond 表达式的规则，它们是代数规则：

```
cond_false： 如果第一个条件是 false：
  (cond                    = (cond
    [false ...]            ; 第一个子句消失。
    [exp1 exp2]            [exp1 exp2]
...)                        ...)
即第一个 cond 子句消失；
cond_true： 如果第一个条件为 true：
  (cond                    = exp
    [true exp]
    ...)
整个 cond 表达式被替换成第一个答案；
cond_else： 如果惟一的子句是 else 子句：
  (cond                   = exp
    [else exp])
```

这个 cond 表达式就被替换成 else 子句中的答案。

考虑如下计算：

```
  (cond
    [false 1]
    [true (+ 1 1)]
    [else 3])

= (cond
    [true (+ 1 1)]
    [else 3])
```

```
= (+ 1 1)

= 2
```

首先，消去一个 cond 子句，然后把 cond 表达式等同于(+ 1 1)，其余的计算就是算术运算了。

如同日常所使用的算术和代数规则，计算规则通常以等式形式给出。事实上，数学上成立的规律同样适用于等式系统。例如，如果 $a=b$ 并且 $b=c$，那么 $a=c$。因此，当能够熟练地进行手工计算后，我们就可以省略掉一些明显的步骤。对于前一个计算过程来说，较为简洁的形式是：

```
  (cond
    [false 1]
    [true (+ 1 1)]
    [else 3])

= (+ 1 1)

= 2
```

更为重要的是，在任何情况下，正如在代数中所做的那样，我们都可以把表达式替换成任何一个与之等值的表达式。下面是另一个 cond 表达式以及它的计算过程：

```
  (cond
    [(= 1 0) 0]
    [else (+ 1 1)])
  ;; 带下划线的表达式先被计算
= (cond
    [false 0]
    [else (+ 1 1)])
  ;; 接着，调用 cond_false 规则
= (cond
    [else (+ 1 1)])
  ;; 使用 cond_else，得到一个算术表达式
= (+ 1 1)
= 2
```

显然，第一步必须计算带下划线的表达式，否则的话，就没有一条可适用的 cond 规则了。当然，这类计算没有任何不寻常，无论是在代数课程中，还是在本书的前几章中，我们都进行过多次这样的计算。

习题

习题 8.3.1　按步计算下列表达式：

```
1. (+ (* (/ 12 8) 2/3)
     (- 20 (sqrt 4)))
2. (cond
       [(= 0 0) false]
       [(> 0 1) (symbol=? 'a 'a)]
       [else (= (/ 1 0) 9)])
3. (cond
       [(= 2 0) false]
```

```
        [(> 2 1) (symbol=? 'a 'a)]
        [else (= (/  1 2) 9)])
```

习题 8.3.2 假设 Definitions 窗口中包含

```
;; f : number number -> number
(define (f x y)
  (+ (* 3 x) (* y y)))
```

说明 DrScheme 如何一步一步地计算下列表达式：

```
1. (+ (f 1 2) (f 2 1))
2. (f 1 (* 2 3))
3. (f (f 1 (* 2 3)) 19)
```

8.4 错　　误

带括号的语句可能是 Scheme 语句，也可能不是，这取决于它们是否符合图 8.2 中的文法。如果 DrScheme 发现某个语句不属于 Beginning Student 语言，就会报告语法错误。

处理后的表达式在语法上是合法的，但是，就我们的计算规则而言，有些语句还会出现问题。我们称这样的表达式包含逻辑错误，或者运行错误。考虑最简单的例子：(/10)。在数学中，计算

$$\frac{1}{0}$$

是没有意义的。因为 Scheme 的计算必须与数学相一致，所以不能把(/10)等同与任何一个值。

一般来说，如果某个表达式没有值，而且计算规则也不允许对该表达式进行进一步的简化，那么就认为出现了一个错误，或者说该函数产生了一个错误消息。实际上，这意味着计算会立即停止，并产生一条错误消息，例如对于除零错误，错误消息就是"/:　divide by zero"。

举例来说，考虑如下计算：

```
  (+ (* 20 2) (/ 1 (- 10 10)))
= (+ 40 (/ 1 0))
= /: divide by zero
```

错误停止了对(/10)上下文(+ 40...)的计算。

要理解程序运行时错误是如何产生错误消息的，我们再一次检查计算规则。考虑下面的函数：

```
;; my-divide : number -> number
(define (my-divide n)
  (cond
    [(= n 0) 'inf]
    [else (/ 1 n)]))
```

现在，假设把 *my-divide* 作用于 0，那么，计算的第一步是：

```
  (my-divide 0)

= (cond
    [(= 0 0) 'inf]
```

```
  [else (/ 1 0)])
```

显然，虽然计算带下划线的表达式会产生错误消息“/: divide by zero”，但现在宣称该函数会产生错误消息还为时过早，毕竟，(= 0 0)为 true，因此该调用可以得出正确的计算结果：

```
(my-divide 0)

= (cond
    [(= 0 0) 'inf]
    [else (/ 1 0)])

= (cond
    [true 'inf]
    [else (/ 1 0)])

= 'inf
```

幸运的是，Scheme 的计算规则也自动考虑了这些情况。我们只需记住何时规则起作用就可以了，例如，在

```
(+ (* 20 2) (/ 20 2))
```

中，加法不能在乘法或者除法之前进行。同样，在

```
(cond
 [(= 0 0) 'inf]
 [else (/ 1 0)])
```

中，带下划线的表达式不会被计算。

作为概括，我们最好记住：

> **表达式计算原则**
>
> 简化最外（最左的）可以计算的表达式。

尽管这条规则很简练，但它总是可以解释 Scheme 的计算结果。

在许多情况下，程序员希望定义能产生错误消息的函数，回忆第 6 章中自带检查的 *area-of-disk* 函数：

```
;; checked-area-of-disk : Scheme-value -> boolean
;; 如果 v 是数的话，计算半径为 v 的圆盘的面积

(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error 'checked-area-of-disk "number expected")]))
```

如果把 *checked-area-of-disk* 作用于一个符号，所得的计算过程会是：

```
(- (checked-area-of-disk 'a)
   (checked-area-of-disk 10))

= (- (cond
       [(number? 'a) (area-of-disk 'a)]
       [else (error 'checked-area-of-disk "number expected")])
```

```
      (checked-area-of-disk 10))

  = (- (cond
        [false (area-of-disk 'a)]
        [else (error 'checked-area-of-disk "number expected")])
      (checked-area-of-disk 10))

  = (- (error 'checked-area-of-disk "number expected")
      (checked-area-of-disk 10))

  = checked-area-of-disk : number expected
```

换一种说法，error 表达式的计算结果和除零一样。

8.5 布尔值表达式

当前定义的 Beginning Student Scheme 语言忽略了两种形式的表达式：and 表达式和 or 表达式。下面把它们加到语言中，这也是一个学习新的语言结构的机会。首先我们必须理解它们的语法、语义以及语用。

这是修改后的文法：

```
<exp>   =    (and <exp> <exp>)
           | (or <exp> <exp>)
```

该文法表明，and 和 or 都是关键字，它们后面都跟着两个表达式。初看，这两个表达式很像函数调用。要理解为什么它们不是函数调用，必须先研究两种表达式的语用。

假设需要给出一种条件，判断 *n* 的倒数是不是 *m*：

```
(and (not (= n 0))
     (= (/ 1 n) m))
```

因为不希望出现意外的除 0 运算，我们把条件表达为两个布尔表达式的 and 形式。接下来，假设在计算时，*n* 就是 0，那么，表达式变成

```
(and (not (= 0 0))
     (= (/ 1 0) m))
```

现在，如果 and 是一个普通的表达式，我们就必须计算它的两个子表达式，而计算第二个子表达式就会产生错误。因此，and 并不是一个基本操作，而是一个特殊的表达式。简而言之，我们用 and 及 or 组合布尔表达式，从而简化计算过程。

一旦理解了 and 和 or 表达式的计算过程，就很容易给出相应的计算规则。而更好的方法是，给出与它们等价的表达式：

```
(and <exp-1> <exp-2>)
≡
(cond
  [<exp-1> <exp-2>]
  [else false])
```

及

```
(or <exp-1> <exp-2>)
=
(cond
  [<exp-1> true]
  [else <exp-2>])
```

这两个等式简化了真正发生在 DrScheme 内部的计算过程，不失为一个很好的模式。

8.6 变量定义

虽然我们的第一个文法没有包括变量定义，但程序不仅仅包括函数定义，还包含变量定义。

下面是变量定义的文法规则：

```
<def>    =    (define <var> <exp>)
```

变量定义的外形类似于函数定义，也是由“(”、关键字 define、变量、表达式以及与第一个括号所对应的“)”组成。关键字 define 把变量定义与其他的表达式相区分，但它并没有与函数定义相区分。要区分这两者，必须观察定义的第二个成分。

接下来，必须了解变量定义的含义是什么，一个类似于

```
(define RADIUS 5)
```

的变量定义只有一个意思，即，在计算中，只要遇到 *RADIUS*，就把它替换成 5。

当 DrScheme 遇到一个定义时，如果定义的右部是一个正确的表达式，就必须先计算该表达式。例如，定义

```
(define DIAMETER (* 2 RADIUS))
```

的右部是表达式(* **2** *RADIUS*)。因为 *RADIUS* 代表了 5，所以这个表达式的值是 10。因此，可以认为整个定义就是

```
(define DIAMETER 10)
```

简而言之，当 DrScheme 遇到变量定义的时候，它先求出右部的值。在计算右部的时候，DrScheme 会使用在该定义之前的所有定义，而不会使用在它之后的定义。一旦 DrScheme 得出了右部的值，它就会记住左部变量所代表的值。以后，在计算（其他）表达式时，所有已被定义的变量都会被它的值所替换。

```
(define RADIUS 10)

(define DIAMETER (* 2 RADIUS))

;; area : number -> number
;; 计算半径为 r 的圆盘的面积
(define (area r)
  (* 3.14 (* r r)))

(define AREA-OF-RADIUS (area RADIUS))
```

图 8.4　一个变量定义的例子

考虑图 8.4 中的定义序列。在 DrScheme 处理该定义序列的过程中，它先确定 *RADIUS* 代表 **10**，

DIAMETER 代表 20，而 *area* 是一个函数名。最终，DrScheme 计算出(*area RADIUS*)为 314.0 并把该值赋给 *AREA-OF-RADIUS*。

习题

习题 8.6.1 构造 5 个变量定义的例子。在定义的右部，分别使用常量和表达式。

习题 8.6.2 手工计算如下定义序列：

```
(define RADIUS 10)
(define DIAMETER (* 2 RADIUS))
(define CIRCUMFERENCE (* 3.14 DIAMETER))
```

习题 8.6.3 手工计算如下定义序列：

```
(define PRICE 5)
(define SALES-TAX (* .08 PRICE))
(define TOTAL (+ PRICE SALES-TAX)
```

8.7 结构体的定义

本节讨论结构体 define-struct 的语法和语义。在定义结构体的时候，实际上定义了好几个基本操作，包括一个构造器，若干个选择器以及一个谓词。因此，define-struct 是目前为止最复杂的 Scheme 结构体。

结构体定义是第三种定义形式，关键字 define-struct 把这种定义形式与函数和变量定义相区分，该关键字后应该有一个名字和一个带括号的名字序列：

```
<def> = (define-struct <var0> (<var-1> ... <var-n>)) .
```

下面是一个简单的例子：

```
(define-struct point (x y z))
```

因为 *point*、*x*、*y* 和 *z* 都是变量，而且括号的位置也符合文法，所以这是一个正确的结构体定义。反之，以下两个带括号的语句：

```
(define-struct (point x y z))
(define-struct point x y z)
```

都不是正确的定义，因为跟在 define-struct 后的不是一个变量名和一个带括号的变量序列。

每个 define-struct 定义都引入了若干个新的基本操作。这些基本操作的名字是与被定义的结构体名相关。假设某个结构体的定义为：

```
(define-struct c (s-1 ... s-n))
```

那么 Scheme 就会引入下列基本操作：

1. **make-c**：构造器；
2. **c-s-1 ... c-s-n**：一系列的选择器；
3. **?**：谓词。

这些操作的地位与+、-、*等一样。因为 define-struct 的用途是引入一类新的值，因此在理解这些新操作的规则之前，我们先回过头来看看值的定义。

简单地说，值的集合不仅仅包括常量，还包括结构体。所谓结构体，就是多个值的复合物。就文法而言，我们必须为每一个 define-struct 添加一个子句：

```
<val>           =          (make-c <val> ... <val>)
```

观察 *points* 结构体，它包含了三个名称，因此，如果 *u*、*v* 和 *w* 是值，(make-point *u v w*)也是值。

现在，我们能够理解这些新操作的计算规则了。如果把 **c-s-1** 作用于某个 *c* 结构体，返回值就是该值的第一个成分。同理，第二个选择器提取第二个成分，第三个选择器提取第三个成分，以此类推。要描述新的数据构造器和选择器之间的关系，最好的方法是使用 *n* 个等式：

```
(c-s-1 (make-c V-1 ... V-n)) = V-1
              ⋮
(c-s-n (make-c V-1 ... V-n)) = V-n
```

其中 *V-1* ... *V-n* 是和 *s-1* ... *s-n* 一样长的值的序列。

就例子而言，这些等式是：

```
(point-x (make-point V U W)) = V
(point-y (make-point V U W)) = U
(point-z (make-point V U W)) = W
```

具体来说，(point-y (make-point 3 4 5))等于 4，而(point-x (make-point (make-point 1 2 3) 4 5))等于(make-point 1 2 3)，因为(make-point 1 2 3)也是值。

谓词 c?可被作用于任何值。如果该值是 *c* 类型的，谓词就返回 true，否则，返回 false。我们可以把这两条规则都转化成等式。第一个等式是：

```
(c? (make-c V-1 ... V-n)) = true
```

它把 **c?**与由 make-c 构造的结构体关联起来；第二个等式是：

```
(c? V) = false; 如果 V 不是由 make-c 构造的值，
```

它把 **c?**和所有其他的值关联起来。

按照惯例，理解等式最好的方法是使用例子，如：

```
(point? (make-point V U W)) = true
(point? U) = false ; 如果 U 是值，但不是 point 结构体的值。
```

所以，(point? (make-point 3 4 5))为 true，而(point? 3)为 false。

习题

习题 8.7.1 判断下列语句是否合法：

```
1. (define-struct personnel-record (name salary dob ssn))
2. (define-struct oops ())
3. (define-struct child (dob date (- date dob)))
4. (define-struct (child person) (dob date))
5. (define-struct child (parents dob date))
```

请解释为什么某些语句是合法的结构体定义；如果某个语句是不合法的结构体定义，也请说明理由。

习题 8.7.2 以下哪些是值？

1. (make-point 1 2 3)
2. (make-point (make-point 1 2 3) 4 5)
3. (make-point (+ 1 2) 3 4)

习题 8.7.3 假设 Definitions 窗口中包含如下语句：

```
(define-struct ball (x y speed-x speed-y))
```

求出下列表达式的计算结果：

```
1. (number? (make-ball 1 2 3 4))
2. (ball-speed-y (make-ball (+ 1 2) (+ 3 3) 2 3))
3. (ball-y (make-ball (+ 1 2) (+ 3 3) 2 3))
```

再观察 DrScheme 是如何处理下列表达式的：

```
1. (number? (make-ball 1 3 4))
2. (ball-x (make-posn 1 2))
3. (ball-speed-y 5)
```

用 DrScheme 来验证你的解答。

图 8.5 是 Beginning Student Scheme 的完整文法。

<def>	=	**(define** (*<var>* *<var>* ...*<var>*) *<exp>*)
		| (**define** *<var>* *<exp>*)
		| (**define-struct** *<var0>* (*<var-1>* ...*<var-n>*))
<exp>	=	*<var>*
		| *<con>*
		| (*<prm>* *<exp>* ...*<exp>*)
		| (*<var>* *<exp>* ...*<exp>*)
		| (**cond** (*<exp>* *<exp>*) ...(*<exp>* *<exp>*))
		|| (**cond** (*<exp>* *<exp>*) ...(else *<exp>*))
		| (**and** *<exp>* *<exp>*)
		| (**or** <exp> <exp>)

图 8.5　Beginning Student Scheme 的完整文法

第二部分

任意数目数据的处理



第9章 复合数据类型之二：表

结构体是表达复合信息的一种方法，当知道有多少个数据应当放在一起时相当有用。但在许多情况下，我们并不知道有多少个数据要放在一起，这时可以使用表，表的长度是任意的，换句话说，表可以表示任意（但必须是有限）数目的数据。

把数据组成表是每个人都会做的事：去杂货店前，把所有想买的东西列成表；安排计划时，把所有要做的事情列成表；每年十二月，小孩会把圣诞节愿望列成表；为了准备一个晚会，我们列出所有要邀请的人。总而言之，在生活中，我们经常使用表来列出信息，Scheme 也使用表来组织数据。在这一章，我们先学习创建表，然后学习设计以表为参数的函数。

9.1 表

在 Scheme 中，

```
empty
```

表示一个空表，使用操作 **cons** 可以从一个空表构造出另一个更长的表，如：

```
(cons 'Mercury empty)
```

这个例子从 empty 表和符号'Mercury 构造一个表。图 9.1 用类似于表示结构体的方式来表示这个表。代表 cons 的方框有两个字段：first 和 rest。在这个特定的例子中，first 字段是'Mercury，rest 字段是 empty。

一旦有了包含一个元素的表，接着就可以使用 cons 构造包含两个元素的表：

```
(cons 'Venus (cons 'Mercury empty))
```

图 9.1 中的第 2 行给出了第 2 个表的图形表示，它也是含有两个字段的方框，但是这次的 **rest** 字段中包含一个方框，实际上就是包含第 1 行中的方框。

最后构造一个含有三个元素的表：

```
(cons 'Earth (cons 'Venus (cons 'Mercury empty)))
```

图 9.1 中的第 3 行显示了有三个元素的表，它的 rest 字段包含一个包含方框的方框。以此类推，如果要构造更长的表，只需不断地把方框放入方框之中，这就像一套中国式礼盒，或者是一组嵌套的酒杯，唯一的差别是，在 Scheme 中，嵌套可以反复不断，不是任何艺术品可比的。

```
(cons 'Mercury empty)
```

first	*rest*
'Mercury	empty

```
(cons 'Venus
  (cons 'Mercury empty))
```

first	*rest*
'Venus	*first* 'Mercury / *rest* empty

```
(cons 'Earth
  (cons 'Venus
    (cons 'Mercury empty)))
```

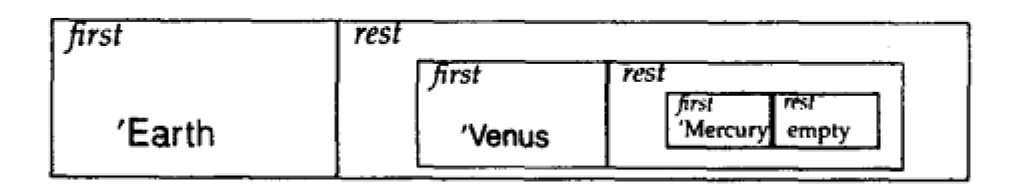


图 9.1 表的图形表示

习题

习题 9.1.1 创建表示以下对象的 Scheme 表

1. 太阳系中的所有行星；
2. 早餐菜谱：牛排、蚕豆、面包、水、果汁、白乳酪和冰淇淋；
3. 基本颜色。

请用类似于图 9.1 的图形表示这些表。

表也可以由数构成，和以前一样，**empty** 仍然表示不含任何东西的表，这是一个含有 10 个数的表：

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9 empty))))))))))
```

该表含有 10 个 cons 和 1 个 empty 表。

表并非一定要由同种类型的值组成，它可以包含任何类型的值，如：

```
(cons 'RobbyRound
  (cons 3
    (cons true
      empty)))
```

在这个表中，第 1 个元素是一个符号，第 2 个元素是一个数，而最后一个元素是一个布尔值[1]。可以认为这个表是一条雇员记录，包含了雇员的名字、在公司工作的年数以及公司是否为该雇员投了健康保险等信息。

假设有一个数表，现在想把表中所有的数加起来。以具体例子进行说明，假定一个表包含三个数，即：

> *list-of-3-numbers*（包含三个数的表）是
>
> (cons *x* (cons *y* (cons *z* empty)))
>
> 其中，*x*、*y* 和 *z* 是数。

跟以前一样，先写出合约、用途说明、函数头部和例子：

```
;; add-up-3 : list-of-3-numbers -> number
;; 求表 a-list-of-3-numbers 中 3 个数之和
;; 例子和测试：
;; (= (add-up-3 (cons 2 (cons 1 (cons 3 empty)))) 6)
;; (= (add-up-3 (cons 0 (cons 1 (cons 0 empty)))) 1)
(define (add-up-3 a-list-of-3-numbers) ...)
```

不过，定义主体时出现了问题。表与结构体很相似，因此下面应该设计包含选择器表达式的模板，然而，我们还不知道如何把元素从表中提取出来。

与结构体选择器类似，Scheme 提供了从表中提取字段的操作：first 和 rest1。first 提取出用 cons 构造表时所用的元素，即第一个字段；rest 提取构造表时的第二个字段。

下面使用等式描述 first、rest 和 cons 之间的关系：

```
  (first (cons 10 empty))
= 10
  (rest (cons 10 empty))
= empty
  (first (rest (cons 10 (cons 22 empty))))
= (first (cons 22 empty))
= 22
```

最后一个等式示范了嵌套表达式的计算，与算术一样，计算从最内层开始，而关键是把(cons *a-value a-list*)看作一个值。在上述计算中，下划线标出了下一步将被简化的表达式。

使用 first 和 rest，可以写出 *add-up-3* 的模板：

```
;; add-up-3 : list-of-3-numbers -> number
;; 求表 a-list-of-3-numbers 中 3 个数之和
(define (add-up-3 a-list-of-3-numbers)
  ... (first a-list-of-3-numbers) ...
  ... (first (rest a-list-of-3-numbers)) ...
```

[1] 传统的名称是 lar 和 cdr，但是我们不使用这些无意义的名称。

```
... (first (rest (rest a-list-of-3-numbers))) ... )
```

这三个表达式说明我们可以分别提取输入的 *a-list-of-3-numbers* 包含的三个成分。

习题

习题 9.1.2 设 *l* 是一个表

```
(cons 10 (cons 20 (cons 5 empty)))
```

请问下列表达式的值分别是什么？

```
1. (rest l)
2. (first (rest l))
3. (rest (rest l))
4. (first (rest (rest l)))
5. (rest (rest (rest l)))
```

习题 9.1.3 完成 *add-up-3* 的设计：先定义主体，然后用一些例子进行测试。

由 3 个数组成的表可以表示三维空间中的一个点。计算三维空间中某个点到原点的距离的方法与二维空间没什么两样，即把所有坐标自乘，相加，再求和的平方根。

使用 *add-up-3* 的模板设计函数 *distance-to-0-for-3*，该函数计算三维空间中某个点到原点的距离。

习题 9.1.4 给出由两个符号组成的表的数据定义，再设计函数 *contains-2-doll?*，该函数读入包含两个符号的表，判断两个符号中是不是有一个是'doll。

cons 和结构体的确切关系：从 cons、first 和 rest 之间关系的讨论中可以看到，cons 实际上是一种结构体，它的两个选择器分别是 first 和 rest：

```
(define-struct pair (left right))
  (define (our-cons a-value a-list) (make-pair a-value a-list))
  (define (our-first a-pair) (pair-left a-pair))
  (define (our-rest a-pair) (pair-right a-pair))
  (define (our-cons? x) (pair? x))
```

尽管上述定义是对 cons 的近似，但它们并不精确。DrScheme 提供的 cons 实际上是 make-pai 自带检查的版本，准确地说，cons 操作确保 right 字段总是表，即要么是 cons 结构，要么是 empty。下面是改进后的定义：

```
(define (our-cons a-value a-list)
  (cond
    [(empty? a-list) (make-pair any a-list)]
    [(our-cons? a-list) (make-pair any a-list)]
[else (error 'cons "list as second argument expected")]))
```

这样，*our-first*、*our-rest*、*our-cons?* 就分别与 first、rest、cons 完全对应了。最后，还必须确保不直接使用 make-pair 来构造表，否则一不小心就会出错。

9.2 任意长的表的数据定义

假设一个出售洋娃娃、化妆品、小丑、弓、箭、足球等各种各样玩具的商店要建立一份货物库存清单。店主可以从空表开始，逐一加入不同的玩具名称。

用 Scheme 表示这样的表非常简单。方法是使用符号代表玩具，然后用 cons 把它们组成表。下面是一些简短的例子：

```
empty
(cons 'ball empty)
(cons 'arrow (cons 'ball empty))
(cons 'clown empty)
(cons 'bow (cons 'arrow (cons 'ball empty)))
(cons 'clown (cons 'bow (cons 'arrow (cons 'ball empty))))
```

对一个真正的商店来说，这个表可能会包含很多物品，而且，随着时间推移，该表还会不断变长缩短。因此无论如何，我们都无法提前知道这个表会含有多少个元素。所以，如果要开发一个函数，以这样的表为参数，就不能简单假定该表中会有 1 个、2 个、3 个还是 4 个元素，而必须准备处理任意长的表。

换一种说法，我们需要一种数据定义，它能够描述包含任意多个符号的表。不幸的是，到目前为止我们所接触到的数据定义只能描述固定大小的数据，如，有着固定组成成分的结构体，或者是固定长度的表。那么我们应该如何描述任意长的表呢？

观察一下前面提到的例子，我们会发现它们属于两个种类。我们从空表开始，使用 cons 构造越来越长的表，每一次使用 cons，都把一个玩具和一个已有的表组成新的表。下面就是描述这个过程的数据定义：

list-of-symbols（符号表）是下列两者之一：

1. 空表 empty。
2. (cons *s* *los*)，其中 *s* 是符号，而 *los* 是由符号组成的表。

这个定义与到目前为止我们遇到过的定义或在中学所学到的所有定义都不同。那些定义都使用已有的、已经被充分理解的概念定义新的概念。与此不同，这个定义在标号为 2 的条款中引用了自己，也就是说，它用符号表来解释什么是符号表。我们称这种类型的定义为自引用或递归。

乍看上去，引用自己来说明自己的定义是没有意义的,但这第一印象是错误的。只要能构造出对象，递归定义就是有意义的；如果能用递归定义构造出所有预期的对象，定义就是正确的[1]。

让我们来检查一下上述定义有没有意义，能不能构造出所需要的对象。从定义的第一个条款，我们知道 empty 是一个符号表；从第二个条款，我们知道可以使用 cons 由一个符号和一个符号表构造更长的表。我们已经知道 empty 是符号表，又知道'doll 是符号，因此(cons 'ball empty)就是符号表，这里的'doll 并没有什么特别之处，在构造符号表的过程中，可以使用任何其他的符号，如：

```
(cons 'make-up-set empty)
(cons 'water-gun empty)
...
```

一旦拥有包含一个元素的符号表，接着就可以用同样的方法构造包含两个元素的表：

```
(cons 'Barbie (cons 'robot empty))
(cons 'make-up-set (cons 'water-gun empty))
(cons 'ball (cons 'arrow empty))
...
```

由此可以得到任意长度的表。显然，该方法可以描述任何玩具库存清单。

习题

[1] 一个数据定义描述一类不只包含预期元素的数据这很常见。这一限制是内在的，并且只是计算限制的许多表现中的一种。

习题 9.2.1 证明所有本节提到的库存清单表都属于类型 *list-of-symbols*。

习题 9.2.2 是否所有含有两个符号的表都属于类型 *list-of-symbols*？请给出证据。

习题 9.2.3 布尔表是一个由布尔值组成的任意长度的表，请给出布尔表的数据定义。

9.3 处理任意长的表

假定一个玩具商店要把货物库存清单存放在计算机之中，这样，店里的员工就可以快速判断商店里是否还有某种玩具存货。简而言之，商店需要一个能够检查库存是否含有玩具'doll 的函数 *contains-doll?*，翻译成 Scheme 的术语就是，判断符号表中是否存在值为'doll 的元素。

有了函数 *contains-doll?*输入数据的定义，接着就要给出函数合约、头部和用途说明：

```
;; contains-doll? : list-of-symbols -> boolean
;; 判断符号'doll 是否存在于 a-list-of-symbols 之中
(define (contains-doll? a-list-of-symbols) ...)
```

按照设计诀窍，下一步应构造一些能够说明 *contains-doll?*的例子。首先，要给出的是对应最简单的输入（即 empty）的函数输出。既然 empty 不包含任何符号，当然也不包含'doll，所以输出应当是 false：

```
(boolean=? (contains-doll? empty)
           false)
```

接着，考虑只包含一个元素的表，下面是两个例子：

```
(boolean=? (contains-doll? (cons 'ball empty))
           false)
(boolean=? (contains-doll? (cons 'doll empty))
           true)
```

在第一个例子中，表中唯一的元素不是'doll，因此输出是 false；在第二个例子中，表中唯一的元素就是'doll，所以输出应该是 true。最后，我们给出两个更为一般的例子，例中每个表都含有多个元素：

```
(boolean=? (contains-doll? (cons 'bow (cons 'ax (cons 'ball empty))))
           false)
(boolean=? (contains-doll? (cons 'arrow (cons 'doll (cons 'ball empty))))
           true)
```

在第一个例子中，输出仍然是 false，这是因为该表不含'doll；在第二个例子中，输出是 true 因为该表含有元素'doll。

下一步应该是设计与数据定义相符的函数模板。既然符号表的数据定义含有两个子句，主体必然是一个 cond 表达式，用于判断传给函数的表是 empty 还是由 cons 建立的表：

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [(cons? a-list-of-symbols) ...]))
```

在 cond 表达式的第二个子句中，如有不使用(cons? *a-list-of-symbols*)，也可以使用 else。

下面分别研究 cond 表达式的每个子句，然后在模板中逐一填上所需的表达式。回忆一下设计诀窍：如果某个子句的输入类型是复合对象，就应该使用选择器表达式。在这个例子中，empty 不是复合对象。除了 empty，表还可以使用 cons 由一个符号和另一个符号表构成，于是应该在模板中添加(first

a-list-of-symbols)和(rest *a-list-of-symbols*)：

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [else ... (first a-list-of-symbols) ... (rest a-list-of-symbols) ...]))
```

按照混合以及复合数据的设计诀窍，得到模板之后，接着就应该考虑在主体内分别处理 cond 的每一个子句。如果(empty? *a-list-of-symbols*)为真，输入就是一个空表，这时函数就应该返回结果 false；如果(cons? *a-list-of-symbols*)为真，按照模板中的注释，该表含有两个部分，它们分别是表的第一个符号以及其余符号组成的表。考虑一个该类型的表：

```
(cons 'arrow
  (cons...
      ...empty)))
```

与人一样，函数也必须先将表中第一个元素与'doll 进行比较。在这个例子中，表的第一个元素是'arrow，所以比较的结果是 false。再考虑另一个例子，比如说：

```
(cons 'doll
  (cons...
   ...empty)))
```

此时输入表的第一个元素是'doll，所以函数应该返回 true。这意味着 cond 表达式的第二个子句还应当包含另一个 cond 表达式：

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [else (cond
            [(symbol=? (first a-list-of-symbols) 'doll)
             true]
            [else
... (rest a-list-of-symbols) ...])]))
```

容易看出，如果'doll 与(first *a-list-of-symbols*)比较的结果为 true，函数也应当返回 true；如果比较的结果为 false，则还要处理另一个符号表，即(rest *a-list-of-symbols*)。换句话说，如果表的第一个元素不是'doll，我们还需要检查表的其余部分是否包含'doll。

幸运的是，我们正好有一个这样的函数：*contains-doll?*，按照其用途说明，它能判断某个表是否包含'doll。*contains-doll?*的用途说明表明，如果 *l* 是符号表，那么(*contains-doll? l*)就会告诉我们 *l* 是否含有符号'doll。与此类似，(*contains-doll?* (rest *l*))能够判断 *l* 的 rest 部分中是否含有符号'doll。按照同样的推理，(*contains-doll?* (rest *a-list-of-symbols*))判断符号'doll 是否存在于(rest *a-list-of-symbols*)之中，而那正是我们所需要的。

下面是完整的函数定义：

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [else (cond
            [(symbol=? (first a-list-of-symbols) 'doll) true]
[else (contains-doll? (rest a-list-of-symbols))])]))
```

该函数读入一个符号表，先判断它是不是空表。如果是的，函数就返回 false。否则，表非空，那么函数返回的结果就取决于表的第一个元素。如果这个元素是'doll，结果就是 true；如果不是，函数的返回值就是检查输入表 rest 部分的结果。

习题

习题 9.3.1 在 DrScheme 中使用下面的例子测试 *contains-doll?* 的定义：

```
empty
(cons 'ball empty)
(cons 'arrow (cons 'doll empty))
(cons 'bow (cons 'arrow (cons 'ball empty)))
```

习题 9.3.2 给出函数 *contains-doll?*第二条 **cond** 子句的另一种表示方法，是将

```
(contains-doll? (rest a-list-of-symbols))
```

理解为一种结果为 **true** 或 **false** 的条件，然后将它与条件

```
(symbol=? (first a-list-of-symbols) 'doll)
```

适当组合起来，请依此重新给出 *contains-doll?*的定义。

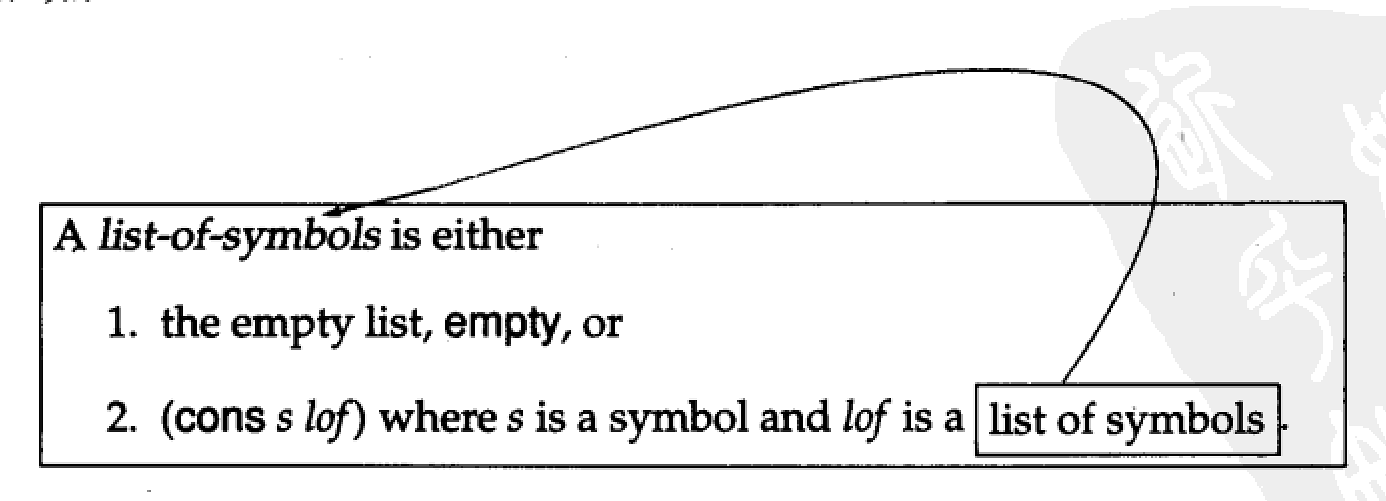
习题 9.3.3 设计函数 *contains?*，该函数读入一个符号和一个符号表，判断符号是否在表中存在。

9.4 设计自引用数据定义的函数

自引用数据定义似乎远比复合数据或混合数据复杂，但是，正如上一节例子所示，以前的设计诀窍仍然适用。不过，这一节，我们要讨论一种新的设计诀窍，它更适合于自引用数据定义，而且概括了复合数据以及混合数据的设计过程。新的设计诀窍着重于发现何时需要自引用数据定义，以及自引用数据定义的模板设计以及主体定义等。

数据分析和设计：若问题描述涉及任意长的复合信息，就需要使用递归或者是自引用数据定义。到目前为止，我们只碰到一个例子，即符号表 *list-of-symbols*。在这一部分以及下一部分，我们会遇到更多的例子[1]。

要使递归的数据定义有意义，必须满足两个条件：第一，该定义必须至少含有两条子句；第二，其中至少有一条子句不能引用定义自身。鉴别自引用的一种较好方式就是用箭头明确地把引用和数据定义连接起来，如：

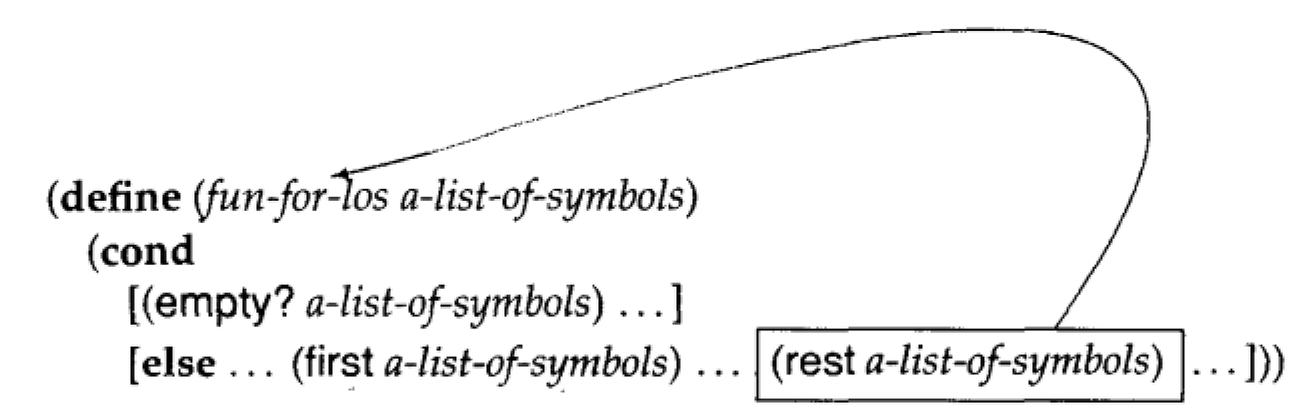


[1] 数似乎也可以是任意大的。对于不精确的数，这是一个错觉。对于精确的整数，确定是这样的。本部分我们将讨论这些数。

模板：自引用的数据定义所描述的是混合数据类型，其中每个子句描述一种子类型。因此，模板的设计可以按照 6.5 节和 7.2 节给出的步骤进行。特别地，每个条件子句与一种数据定义相对应，因此 cond 表达式的子句数应该和数据定义的子句数一样。

另外，检查每一个选择器表达式，如果某个选择器表达式的返回值类型与函数的输入数据类型一致，就用箭头把它和函数的参数连接起来。最后得到的箭头数目必然与数据定义中的箭头数目相同。

下面是表处理函数的模板，它包含了两个子句和一个箭头：



为了简单起见，本书将使用文字代替箭头，方法是把函数本身作用于选择器表达式，从而调用自己：

```
(define (fun-for-los a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols)...]
    [else...(first a-list-of-symbols)...
...(fun-for-los (rest a-list-of-symbols))...]))
```

这种类型的自调用一般称为自然递归。

主体：设计主体从不包含自然递归的那些 cond 子句开始。这些子句被称为基本情况，它们所对应的答案一般已由例子给出，或者很容易得到。

然后再来处理那些包含自引用的情况。首先，考虑模板中的每一个表达式计算什么，对于那些递归调用，我们假设函数已经能够按照我们指定的用途说明工作，剩下的问题就是把不同的值结合起来。

假设我们要定义函数 *how-many*，该函数求出一个符号表中包含多少个符号。按照设计诀窍，有：

```
;; how-many : list-of-symbols -> number
;; 求出 a-list-of-symbols 中包含多少个符号
(define (how-many a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [else ... (first a-list-of-symbols) ... [1]
... (how-many (rest a-list-of-symbols)) ...]))
```

对于基本情况，因为空表不包含任何东西，答案是 0；而第二个子句中的两个表达式分别给出表的元素以及(**rest** *a-list-of-symbols*)所包含的符号数。要计算出 *a-list-of-symbols* 中包含了多少个符号，只需在后一个表达式的值上加 1：

```
(define (how-many a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) 0]
    [else (+ (how-many (rest a-list-of-symbols)) 1)]))
```

把值结合起来：在许多情况下，可以使用 Scheme 的基本操作，如+、and 或 cons。如果问题描述包

[1] 数似乎也可以是任意大的，对于不精确的数，这是一个错觉。对于精确的整数，确定是这样的。本部分我们将讨论这些数。

含了对第一个元素的处理说明，我们可能还需要使用嵌套的 cond 语句。最后，在某些情况下，可能还需要定义辅助函数。

图 9.2 总结了上述讨论，那些没有讨论过的设计诀窍仍然和以前一样给出。下一节我们将详细讨论一些例子。

阶段	目标	活动
数据分析和设计	阐明数据定义	设计数据定义，至少考虑两种情况： • 不引用定义。 • 在数据定义中显式标明所有的自引用
合约、用途说明和函数头部	给函数命名； 说明输入和输出数据的类型； 描述函数的用途； 阐明函数头部	给函数命名、说明输入数据和输出数据的类型，指出函数的用途： ;; *name* : *in1 in2* ...--> *out* ;; 从 *x1* ..计算 ... (define (*name x1 x2* ...) ...)
例子	使用例子刻划输入和输出间的关系	创建刻划输入输出关系的例子： • 确信对于每种子类都至少有一个例子
模板	阐明程序框架	对于每种可能情况都设计一个 **cond** 表达式： • 对于每个子句添加选择器 。 • 将主体标记为递归。 • 测试模板中的自引用是否与数据定义匹配
主体	定义函数	对每个 **cond** 阐明一个 Scheme 表达式： • 按照用途说明阐明每个递归表达式所计算的值
测试	发现错误（拼写错误和逻辑错误）	将函数应用于例子的输入： • 检查程序输出是否与预期的值相符

图 9.2 处理自引用数据的函数设计

9.5 更多关于简单表的例子

我们从价格着手再次考虑货物库存清单。除了货物表之外，店主还应当有一张物品价格表。有了价格表，店主就可以知道现在所有的玩具值多少钱，或者，比较年初的库存清单和年末的库存清单，店主就可以计算出一年的盈利是多少。

价格表可以用一个表来表示，例如：

```
empty
(cons 1.22 empty)
(cons 2.59 empty)
(cons 1.22 (cons 2.59 empty))
(cons 17.05 (cons 1.22 (cons 2.59 empty)))
```

对于一个商店来说，我们仍然不能给这样的表加上长度限制，并且所有处理这张表的函数都必须准备读入任意长的表。

假设玩具店现在需要一个函数，从每一件玩具的价格计算出所有玩具的总价。我们把这个函数称为 *sum*。在定义 *sum* 之前，我们必须先解决如何描述函数输入的问题。显然，函数的输入是数值组成的表。简而言之，我们需要这样一种数据定义，它能精确地定义任意长的数表。事实上，通过把符号表定义中的“symbol”替换成“number”，就可以得到这样的定义：

> *list-of-numbers*（数表）是下列两者之一：
> 1. 空表 empty。
> 2. (cons *n lon*)，其中 n 是数，而 *lon* 是由数组成的表。

跟上例一样，数据定义是自引用的。我们先证实它的确定义了一个表，并且定义的是我们所希望的表。前面所列 出的都是数表，其中第 1 个表 empty 显然属于该数据定义，第 2 个和第 3 个表分别是把数 1.22 和 2.59 用 cons 连接到空表上得到的，其他表也是使用类似方法得到的。

按照惯例，我们从合约、头部和用途说明开始设计函数：

```
;; sum : list-of-numbers -> number
;; 计算 a-list-of-nums 中数的总和
(define (sum a-list-of-nums) ...)
```

接着来看该函数的一些例子：

```
(= (sum empty)
   0)
(= (sum (cons 1.00 empty))
   1.0)
(= (sum (cons 17.05 (cons 1.22 (cons 2.59 empty))))
   20.86)
```

如果 *sum* 被作用于 empty，依题意商店中没有任何库存，因此结果是 0。如果输入是(cons 1.00 empty)，即商店中只有一种玩具，那么所有玩具的价格总和就是这个玩具的价格，所以结果是 1.00。最后，对于 (cons 17.05 (cons 1.22 (cons 2.59 empty)))，*sum* 应当返回：

```
17.05+1.22+2.59=20.86
```

下面我们按照设计诀窍一步一步来设计 *sum* 的模板。第一步，添上 cond 表达式：

```
(define (sum a-list-of-nums)
  (cond
    [(empty? a-list-of-nums) ...]
[(cons? a-list-of-nums) ...]))
```

其中第二个子句表明它是用来处理由 cons 构造的表的。第二步，为每个子句添上合适的选择器表达式：

```
(define (sum a-list-of-nums)
  (cond
    [(empty? a-list-of-nums) ...]
    [(cons? a-list-of-nums)
     ...(first a-list-of-nums) ... (rest a-list-of-nums) ...]))
```

最后一步，添上 *sum* 的自然递归，也就是处理数据定义中的自引用部分：

```
(define (sum a-list-of-nums)
  (cond
    [(empty? a-list-of-nums) ...]
    [else ... (first a-list-of-nums) ... (sum (rest a-list-of-nums)) ...]))
```

最终的模板几乎已经包含了数据定义的每一个方面：两个子句。第二个子句中的 cons 结构，以及第二个子句中的自引用。数据定义中唯一没有在函数模板中得到反映的部分是，cons 结构的第一个部分是数。

既然已经有了模板，下面就一个子句一个子句地定义 cond 表达式。在第一个子句中，输入是 empty，

表示商店没有库存，在这种情况下答案是0；在模板的第二个子句中，我们有两个表达式：

1. (**first** *a-list-of-nums*)，它提取第一件玩具的价格；
2. (*sum* (**rest** *a-list-of-nums*))，按照 *sum* 的用途说明，它的计算结果是(**rest** *a-list-of-nums*)中玩具价格的总和。

从这两个表达式中，我们发现表达式：

```
(+ (first a-list-of-nums) (sum (rest a-list-of-nums)))
```

正好计算出第二个 cond 子句的答案。

函数 *sum* 的完整定义如下：

```
(define (sum a-list-of-nums)
  (cond
    [(empty? a-list-of-nums) 0]
    [else (+ (first a-list-of-nums) (sum (rest a-list-of-nums)))]))
```

对函数定义、模板和数据定义三者进行比较，可以看出，从数据定义到模板是设计函数过程中的主要步骤。通过对输入的认识，得出函数模板，一旦得到了模板，就可以把值结合起来。对于简单的例子，这个步骤很简单；对于复杂的例子，它就需要严密的思考了。

在以后的章节中，我们会了解到，数据定义和函数这种外在关系并不是偶然的，函数输入的数据定义往往在很大程度上决定函数的框架。

习题

习题 9.5.1　在 DrScheme 中，使用下列数表测试 *sum* 的定义：

```
empty
(cons 1.00 empty)
(cons 17.05 (cons 1.22 (cons 2.59 empty)))
```

将结果与手工计算所得进行比较。然后把 *sum* 作用于下列数表：

```
empty
(cons 2.59 empty)
(cons 1.22 (cons 2.59 empty))
```

先手工确定结果是什么，然后使用 DrScheme 进行计算。

习题 9.5.2　设计函数 *how-many-symbols*，该函数读入一个符号表，返回表中元素的数目。设计函数 *how-many-numbers*，该函数读入一个数表，返回表中元素的数目。思考一下 *how-many-symbols* 和 *how-many-numbers* 有什么差别？

习题 9.5.3　设计函数 *dollar-store?*，该函数读入一个物价表，检查是否所有的价格都小于 1。

例如，下列表达式的计算结果应当是 **true**：

```
(dollar-store? empty)
(not (dollar-store? (cons .75 (cons 1.95 (cons .25 empty)))))
(dollar-store? (cons .75 (cons .95 (cons .25 empty))))
```

进一步设计一个更一般的函数，该函数读入一个物价表和限价，检查物价表中所有价格是不是都小于限价。

习题 9.5.4　设计函数 *check-range1*，该函数读入由温度测量值组成的表，检查是否所有的温度值都在 5℃和 95℃之间。把这个函数一般化为 *check-range*，该函数读入由温度测量值组成的表和一个区间，检查是否所有的温度测量值都落在该区间。

习题 9.5.5　设计函数 *convert*，该函数读入一个数表，并返回对应的数。表中的第一个数是数的最

低位，以此类推。

开发函数 *check-guess-for-list*，实现习题 5.1.3 中猜数字游戏的一般版本。该函数读入两个输入：数表 *guess*，代表玩家的猜测；数 *target*，代表随机生成的隐含数。根据数（用 *convert*）转换成的数与 *target* 的关系，输出三种结果中的一种：'TooSmall、'Perfect 或者'TooLarge。

教学包 guess.ss 实现了这个游戏的其余部分。想玩这个游戏的话，请在完成函数的设计之后，使用该教学包并计算下面的表达式：

```
(guess-with-gui-list 5 check-guess-for-list)
```

习题 9.5.6　设计函数 *delta*，该函数读入两个价格表（也就是数表）。第一个表代表月初的库存清单，第二个表代表月末的库存清单。函数的输出是两个价格的差，如果价格上涨了，其值就是正的；如果价格下跌了，其值就是负的。

习题 9.5.7　定义函数 *average-price*，该函数读入一个价格表，并计算玩具的平均价格。平均价格是总的价格除以玩具的数量。

逐步求精：先开发能够处理非空表的函数，然后设计自带检查的函数（参见第 7.5 节），当后者作用于空表时，给出错误信息。

习题 9.5.8　设计函数 *draw-circles*，该函数读入额一个 *posn* 结构体 *p* 和一个数表。表中的每一个数都代表某个圆的半径。该函数使用操作 *draw-circle*，在画布上绘制一系列以 *p* 为圆心的同心圆。如果函数能够画出所有的圆，就返回 *true*；否则，提示错误。

使用教学软件包 draw.ss，并用(*start* 300 300)建立画布。回忆一下，draw.ss 提供了结构体 *posn* 的定义（参见第 7.1 节）。



表的进一步处理



第 9 章讨论的函数可以处理由数、符号和布尔值等原子数据组成的表。函数应该能够生成这样的表，也应当能够处理由结构体组成的表。这一章就来讨论这些情况，同时，进一步研究设计诀窍的使用。

10.1 返回表的函数

回顾 2.3 节中的 *wage* 函数：

```
;; wage : number -> number
;; 计算某个雇员工作 h 小时的总工资（每小时工资 12 美元）
(define (wage h)
  (* 12 h))
```

wage 函数读入某个员工周工作时间，返回他的周工资。为了简单起见，假设所有员工每小时的工资都是一样的，即 12 元。因为 *wage* 只能计算一个人的工资，公司是不会对它感兴趣的，公司需要的是一个能够计算所有员工工资的函数。

我们把这个新的函数叫做 *hours->wages*，该函数读入公司所有员工一周的工作时数，返回所有员工一周应得的工资。显然，函数的输入和输出都可以使用由数组成的表来表示。既然有了输入和输出的数据定义，下面开始函数的设计：

```
;; hours->wages : list-of-numbers -> list-of-numbers
;; 由周工作时间表（alon）创建周工资表
(define (hours->wages alon) ...)
```

接下来是一些输入和输出的例子：

```
empty
(cons 28 empty)
(cons 40 (cons 28 empty))

empty
(cons 336 empty)
(cons 480 (cons 336 empty))
```

输出表是通过计算输入表中每一个元素所对应的工资额得到的。

既然 *hours->wages* 的输入数据类型与 *sum* 函数的一致，而函数的模板仅取决于输入的数据定义，所以我们可以再次利用 *list-of-numbers* 模板：

```
(define (hours->wages alon)
  (cond
```

```
    [(empty? alon) ...]
    [else ... (first alon) ... (hours->wages (rest alon)) ...]))
```

从模板出发，下面开始函数设计中最具有挑战性的一步：定义主体。按照设计诀窍，我们从最简单的子句开始，分别考虑每一个 cond 子句。首先，假设(empty? *alon*)为真，即输入是 empty，这时，输出也是 empty：

```
(define (hours->wages alon)
  (cond
    [(empty? alon) empty]
[else ... (first alon) ... (hours->wages (rest alon)) ...]))
```

其次，假设 *alon* 由一个数和一个数表经 cons 连接而成，这时，设计诀窍要求我们明确说明这两个表达式分别计算出什么：

1. (first *alon*)返回 *alon* 中的第一个数值，即工作时间表中的第一个元素。

2. (*hours->wages* (rest *alon*))提醒我们(rest *alon*)是一个表，并且这个表可以被正在定义的函数处理。按照函数的用途说明，这个表达式会计算出工作时间表其余部分所对应的工资表，虽然还没有完成函数定义，但在设计函数时可以假设这是对的。

到此，函数定义就只剩下一小步了。既然我们已经得到了 *alon* 中除了第一个元素以外全部元素所对应的工资表，函数必须完成如下两件事，才能得出整个工作时间表所对应的周工资表：

1. 计算第一个雇员的工作时间所对应的周工资。

2. 使用第一个雇员工作时间所对应的周工资以及(rest *alon*)所对应的周工资表，构造一个表，该表就是与表 *alon* 对应的所有雇员的周工资组成的表。

第一步可使用 *wage*；对于第二步，可以使用 cons 把这两者连接起来，构成一个表：

```
(cons (wage (first alon)) (hours->wages (rest alon)))
```

这样，就得到了完整的函数。图 10.1 给出了这个函数。

```
;; hours->wages : list-of-numbers -> list-of-numbers
;; 由周工作时间表（alon）创建周工资表
(define (hours->wages alon)
  (cond
    [(empty? alon) empty]
    [else (cons (wage (first alon)) (hours->wages (rest alon)))]))

;; wage : number -> number
;; 计算某个人工作 h 小时的总工资（每小时的工资是 12 元）
(define (wage h)
  (* 12 h))
```

图 10.1　计算周工资

习题

习题 10.1.1　如果将每个人的工资提升为每小时 14 元，请问应如何修改图 10.1 中的函数？

习题 10.1.2　没有人能够每周工作 100 小时以上。为了防止欺骗，*hours->wages* 函数应当对输入进行检查，确保没有一个元素的值超过 100。如果表中某一元素超过了 100，函数应当立即给出错误信息“too many hours”。请问应该如何修改图 10.1 中的函数，使得它能够执行上述真实性检查？

习题 10.1.3　开发函数 *convertFC*，该函数把含华氏温度值的表转换成含摄氏温度值的表。

习题 10.1.4　设计函数 *convert-euro*，基于 1.22 欧元 1 美元的汇率，把美元数值的输入转换成欧元

数值。把 *convert-euro* 一般化为 *convert-euro-1*，该函数读入汇率以及一个美元数额表，使用该汇率把美元数额的表转换成欧元数额的表。

习题 10.1.5 设计函数 *eliminate-exp*，该函数读入数 *ua* 和玩具价格表 *lotp*，结果为 *lotp* 中所有小于等于 *ua* 的元素组成的表。例如：[1]

```
(eliminate-exp 1.0 (cons 2.95 (cons .95 (cons 1.0 (cons 5 empty)))))
;; 预期值:
(cons .95 (cons 1.0 empty))
```

习题 10.1.6 设计函数 *name-robot*，该函数读入一个由玩具名称组成的表，返回一个更精确的玩具名称表，详细说来，就是把表中所有的'robot 替换为'r2d2，其他玩具名称保持不变。

把 *name-robot* 一般化为函数 *substitute*。这个新的函数读入两个符号（分别名为 *new* 和 *old*）以及一个符号表，返回一个新的符号表，其中所有的 *old* 都被替换成 *new*。例如：

```
(substitute 'Barbie 'doll (cons 'robot (cons 'doll (cons 'dress empty))))
;; 预期值:
(cons 'robot (cons 'Barbie (cons 'dress empty)))
```

习题 10.1.7 设计函数 *recall*，从表中去除某些特定的玩具。该函数读入玩具的名字 *ty* 和表 *lon*，返回一个表，该表保留了除 *ty* 以外 *lon* 的所有元素。例如：

```
(recall 'robot (cons 'robot (cons 'doll (cons 'dress empty))))
;; 期望值:
(cons 'doll (cons 'dress empty))
```

习题 10.1.8 设计求解二次方程的函数 *quadratic-roots*（参见习题 4.4.4 和习题 5.1.4），该函数的输入为方程的系数，即 *a*、*b* 以及 *c*，所执行的计算根据输入而定：

1. 如果 a ＝0，输出 **'degenerate**。
2. 如果 $b^2 < 4ac$，二次方程没有解。在这种情况下，*quadratic-roots* 返回 **'none**。
3. 如果 $b^2 = 4ac$，二次方程有一个解：

$$\frac{-b}{2\cdot a};$$

这个解就是函数的答案。

4. 如果 $b^2 > 4ac$，方程有两个解：

$$\frac{-b+\sqrt{b^2-4\cdot a\cdot c}}{2\cdot a}$$

和

$$\frac{-b-\sqrt{b^2-4\cdot a\cdot c}}{2\cdot a};$$

函数的返回值是两个由数组成的表：第一个解后跟着第二个解。

用习题 4.4.4 和习题 5.1.4 中的例子来测试这个函数。先确定每个例子的答案，然后使用 DrScheme 求解。

---

[1] 因为我们还不知道如何用函数来比较两个表，所以还是老式的测试方法。

习题 10.1.9. 在许多杂货店，收银员需要向顾客报出价格。收银员的计算机对于顾客必须支付的金额，构造含有如下五个元素的表：

1. 元的数额；
2. 如果元的数额是 1，这一元素是符号'dollar，否则就是'dollars；
3. 符号'and；
4. 分的数额；
5. 如果分的数额是 1，这一元素是符号'cent，否则就是'cents。

开发函数 *controller*，该函数读入一个数，返回如上描述的表。例如，如果金额数是＄1.03，那么(*controller* 103)的计算过程如下：

```
(controller 103)
;; 预期值：
(cons 1 (cons 'dollar (cons 'and (cons 3 (cons 'cents empty)))))
```

提示：Scheme 提供了算术操作 quotient 和 remainder，对于整数 n 和 m，分别生成 n/m 的商和余数。

如果在 *controller* 返回的表中元的数额和分的数额在 0 至 20 之间的话，请用一台能说话的计算机对它进行测试。教学软件包 sound.ss 提供了两种操作：*speak-word* 和 *speak-list*，前者接受符号或数，后者接受由符号和数值组成的表，它们都能念出读入的参数。通过求诸如(*speak-word* 1)，(*speak-list* (cons 1(cons 'dollar empty)))和 (*speak-list* (cons 'beautiful (cons 'lady empty))) 等表达式的值，了解它们是如何工作的。

挑战：教学包 sound 只包含 0 到 20 以及 30、40、50、60、70、80、90 的数字发音。由于这个限制，现在只能念出数额在 0 到 20 之间的元和分的值。请实现一个能够处理 0 到 99.99 之间任意金额的 *controller* 函数。

10.2　包含结构体的表

用符号表或者价格表来表示库存清单的想法是不现实的。玩具店的销售员不仅要知道玩具的名字和价格，可能还需要了解一些其他的属性，例如库存量、交货时间，甚至于它的照片。类似地，用表来表示员工的周工作时间也不是一个好方法，即使是打印一张支付薪水的支票，也需要其他的信息。

幸好，表中的元素并非一定是原子值。表可以包含任何东西，特别是结构体。让我们试着给出一张更为现实的玩具店库存清单，先从库存记录的结构体和数据定义开始：

```
(define-struct ir (name price))
```

> *inventory-record*(库存记录，简写为 *ir*）是一个结构体：
>
> 　　(make-ir *s n*)
>
> 其中 *s* 是一个符号，*n* 是一个正数。

现在可以定义表示库存清单的表：

> *inventory*（库存清单）是下列两者之一：
>
> 1. empty
> 2. (cons *ir inv*)
>
> 其中 *ir* 是一条库存记录，*inv* 是一个库存清单。

虽然表的定义形态与以前一样，但表的成分的数据定义是分开给出的。既然这是第一次定义这样的数据，在继续学习之前，先构造几个例子。

最简单的库存清单的例子就是 empty。要创建更大的库存清单，必须先创建一个库存记录，然后用 cons 把它和另一个库存清单连接起来：

```
(cons (make-ir 'doll 17.95)
  empty)
```

接着，就可以创建更大的库存清单：

```
(cons (make-ir 'robot 22.05)
 (cons (make-ir 'doll 17.95)
   empty))
```

现在开始改写处理库存清单的函数。首先来看 *sum*，它读入库存清单并返回其价格总和。下面是改写后的该函数的基本信息：

```
;; sum : inventory -> number
;; 计算 an-inv 的价格总和
(define (sum an-inv) ...)
```

对上述三个库存清单，该函数产生的结果应该为：0、17.95 和 40.0。

既然库存清单的数据定义基本上就是表的数据定义，我们可以从表处理函数的模板开始：

```
(define (sum an-inv)
  (cond
    [(empty? an-inv) ...]
    [else ... (first an-inv) ... (sum (rest an-inv)) ...]))
```

按照设计诀窍，模板只反映输入的数据定义，并不反映其成分的数据定义，所以这个 sum 的模板与第 9.5 节中的模板并没有差别。

要定义函数的主体，可分别考虑每一个 cond 子句。首先，如果(empty? *an-inv*)为真，*sum* 应该输出 0，所以，第一个 cond 子句中的答案显然就是 0。

```
(define (sum an-inv)
  (cond
    [(empty? an-inv) 0]
    [else (+ (ir-price (first an-inv)) (sum (rest an-inv)))]))
```

图 10.2 计算库存清单的价格

其次，如果条件(empty? *an-inv*)为假，换句话说，如果将 *sum* 应用于 cons 结构的库存清单上，设计诀窍要求我们理解下列两个表达式的目的：

1. (first an-inv)，它提取出表的第一个元素。
2. (sum (rest an-inv))，它提取出表的其余部分，然后应用 sum 进行计

要计算出整个输入 *an-inv* 表的总价，必须确定表中第一个元素的价格。第一个元素的价格可以用选择器 ir-price 得到，该选择器的功能是从一条库存记录中提取出价格。现在，只需把第一个元素的价格和其余部分的价格加起来：

```
(+ (ir-price (first an-inv))
   (sum (rest an-inv)))
```

图 10.2 给出了完整的函数定义。

习题

习题 10.2.1　改写函数 *contains-doll?*，使其输入为库存清单，而不是符号表：

```
;; contains-doll? : inventory -> boolean
;; 测定 an-inv 是否包含一条'doll 记录
(define (contains-doll? an-inv) ...)
```

同时，改写函数 *contains?*，使其输入为一个符号和一个库存清单，并测定库存清单中是否存在具有这个符号的记录：

```
;; contains? : symbol inventory -> boolean
;; 测定库存清单中是否包含一条 asymbol 记录
(define (contains? asymbol an-inv) ...)
```

符　号	价　格	照　片
robot	29.95	
doll	11.95	
rocket	19.95	
⋮	⋮	⋮

图 10.3　一个玩具表

习题 10.2.2 给出包含每个物品照片的库存清单的数据定义和结构体定义。说明如何表示图 10.3 所示的库存清单。

设计函数 *show-picture*，该函数读入一个符号（玩具的名字）和一个上述定义的库存清单，函数返回相应的玩具照片，如果库存清单中没有此种玩具，则返回 false。

习题 10.2.3 设计函数 *price-of*，该函数读入一个玩具的名字和一个库存清单，并返回该玩具的价格。

习题 10.2.4 通讯录建立了人名和电话号码之间的对应关系。给出电话记录和通讯录的数据定义，然后使用这些数据定义设计以下函数：

1. *whose-number*。给定通讯录以及一个电话号码，它查出对应的人名。
2. *phone-number*。给定通讯录以及一个人名，它查出对应的电话号码。

假设一个商人希望从库存清单中分离出那些售价不到或者等于 1 元的商品，并把它们放在一个分店进行销售。要执行这种划分，这个商人需要一个函数，能够从库存清单表中提取出符合要求的元素，也就是说，一个返回由结构体组成的表的函数。

因为这个函数使用库存清单中所有价格小于等于 1.00 的物品建立一个新的货物清单，我们就把它命名为 *extract1*。这个函数读入一个库存清单，返回符合要求的物品的库存清单，所以很容易得出 *extract1* 的合约：

```
;; extract1 : inventory -> inventory
;; 用 an-inv 中所有售价小于等于 1 元的物品建立一个库存清单
(define (extract1 an-inv) ...)
```

我们仍然可以使用原来的 3 个例子来表明 *extract1* 的输入和输出的关系。不幸的是，对于这 3 个例子，由于所有物品的价格都超过了 1 元，所以该函数的返回值都是空的库存清单。为了得到能够表示输入和输出关系的更有效的例子，我们需要有其他物品的库存清单：

```
(cons (make-ir 'dagger .95)
      (cons (make-ir 'Barbie 17.95)
            (cons (make-ir 'key-chain .55)
                  (cons (make-ir 'robot 22.05)
                        empty))))
```

在这个新的库存清单的四个物品中，有两件的价格不到一元。如果把这个表传给 *extract1*，得到的结果是：

```
(cons (make-ir 'dagger .95)
      (cons (make-ir 'key-chain .55)
       empty))
```

它按照原表的顺序列出了那些符合要求的物品。

函数的合约表明，*extract1* 的模板与 *sum* 的模板是一样的（除了名字不同以外）：

```
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) ...]
    [else ... (first an-inv) ... (extract1 (rest an-inv)) ...]))
```

跟往常一样，*sum* 和 *extract1* 在输出上的区别并不影响模板的设计。

为了定义主体，下面单独分析每种情况。第一种情况，如果(empty? *an-inv*)为真，那么答案显然就是 empty，因为这商店中没有任何价格低于 1 元的物品。第二种情况，如果输入表不空，先要确定相应的 cond 子句中两个表达式计算出的结果。因为 *extract*1 是我们遇到的第一个返回由结构体组成的表的函数，让我们先来研究下面这个有趣的例子：

```
;; extract1 : inventory -> inventory
;; 用 an-inv 中所有售价小于等于 1 的物品建立库存清单
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) empty]
    [else (cond
            [(<= (ir-price (first an-inv)) 1.00)
             (cons (first an-inv) (extract1 (rest an-inv)))]
            [else (extract1 (rest an-inv))])]))
```

图 10.4　从库存清单中提取出不到一元的商品

```
(cons (make-ir 'dagger .95)
      (cons (make-ir 'Barbie 17.95)
            (cons (make-ir 'key-chain .55)
                  (cons (make-ir 'robot 22.05)
                        empty))))
```

如果 *an-inv* 表示下面这个库存清单：

```
(first an-inv) = (make-ir 'dagger .95)
(rest an-inv) = (cons (make-ir 'Barbie 17.95)
                  (cons (make-ir 'key-chain .55)
                        (cons (make-ir 'robot 22.05)
                         empty)))
```

假设 *extract1* 能够正常地工作，则：

```
(extract1 (rest an-inv)) = (cons (make-ir 'key-chain .55)
                              empty)
```

换句话说，*extract1* 的递归调用返回了 *an-inv* 其余部分的正确选择，其中 *an-inv* 是一个有单个库存记录的表。

要得到所有 *an-inv* 的正确选择，必须决定如何处理表的第一个元素，这个元素的价格可能大于 1 元，也可能小于 1 元，这表明对于后一种情况应该使用如下的模板：

```
... (cond
      [(<= (ir-price (first an-inv)) 1.00) ...]
      [else ...]) ...
```

如果第一个元素的价格是 1 元，或者更少，它就应该被包含在最后的输出之中。按照要求，它应当是输出的第一个元素。用 Scheme 语言来说，输出应当是这样的一个表，其第一个元素是(first *an-inv*)，其余部分是递归返回的结果。如果第一个元素的价格多于一元，这个元素就应当被排除，也就是说，返回值应当就是对 *an-inv* 的 rest 部分递归所得的结果。图 10.4 给出了这个函数完整的定义。

习题

习题 10.2.5　定义函数 *extract>1*，该函数读入一个库存清单，用其中所有售价超过一元的物品建

立一个库存清单。

习题 10.2.6 设计 *inventory1* 的精确数据定义，*inventory1* 是价格等于一元的物品的库存清单。使用新的数据定义，*extract1* 的合约是：

```
;; extract1 : inventory -> inventory1
(define (extract1 an-inv) ...)
```

请问新定义的合约是否影响函数的开发？

习题 10.2.7 设计函数 *raise-prices*，该函数读入一个库存清单，返回一个库存清单，其中所有的商品都涨价 5%。

习题 10.2.8 使用新的库存清单定义，修改习题 10.1.7 中的 *recall* 函数。该函数读入玩具的名字 *ty* 和一个库存清单，返回一个库存清单，该清单包含输入中除了名为 *ty* 外的所有元素。

习题 10.2.9 使用新的库存清单的定义，修改习题 10.1.6 中的 *name-robot* 函数。该函数读入一个库存清单，返回一个新的库存清单，其中所有的'robot 被替换成'r2d3。

把 *name-robot* 一般化成函数 *substitute*。新的函数读入名为 *new* 和 *old* 的两个符号及一个库存清单，返回一个新的库存清单，其中 *old* 都被替换成 *new*，其他元素不变。

10.3 补充练习：移动图片

在 6.6 节和 7.4 节中，我们学习了单一图形的移动。不过，图片不是单一的形状，而是图形的集合。考虑到常常要绘制、平移以及清除图形，而且希望同时改变或是管理几个图形，所以最好把一个图片的所有部分都存放在一条数据之中。因为一个图片所包含的形状的数目是不确定的，所以最好用表来表示图片。

习题

习题 10.3.1 给出由图形组成的表的数据定义（习题 7.4.1 给出了 *shapes* 的定义）。假设画布的大小是 300×100，创建脸的图形表，并命名为 *FACE*，其基本尺寸如下：

形状	位置	大小	颜色
Circle	(50,50)	40	red
Rectangle	(30,20)	5 × 5	blue
Rectangle	(65,20)	5 × 5	blue
Rectangle	(40,75)	20 × 10	red
rectangle	(45,35)	10 × 30	blue

设计模板 *fun-for-losh*，即以 *list-of-shapes* 为参数的函数的框架。

习题 10.3.2 使用模板 *fun-for-losh* 设计函数 *draw-losh*，该函数读入表 *list-of-shapes*，绘制表中的每一个元素，并返回 true。请在使用这个函数之前先用(*start n m*)创建画布。

习题 10.3.3 使用模板 *fun-for-losh* 设计函数 *translate-losh*，该函数读入 *list-of-shapes* 以及数 *delta*，返回值是一个图形表，其中每个图形都向 *x* 方向平移了 *delta* 个像素。这个函数对于画布没有任何的影响。

习题 10.3.4 使用模板 *fun-for-losh* 设计函数 *clear-losh*，该函数读入 *list-of-shapes*，从画布上清除表中的每个图形，并返回 true。

习题 10.3.5 设计函数 *draw-and-clear-picture*，该函数读入一个图像。它的任务是绘制图像，然后

等待一段时间，再清除图片。

习题 10.3.6 开发函数 *move-picture*。这个函数读入一个数（*delta*）和一个图片 *picture*。它绘制图片，然后等待一段时间，再清除图片，最后绘制平移后的图片。函数的返回值应当是平移 *delta* 像素后的图片。使用下列表达式测试这个函数：

```
(start 500 100)

(draw-losh
  (move-picture -5
    (move-picture 23
(move-picture 10 FACE))))

(stop)
```

这些表达式分别将 *FACE*（参见习题 10.3.1）沿 x 轴正方向平移 10、23 以及-5 个像素。

测试过这个函数之后，使用教学软件包 arrow.ss 计算下面的表达式：

```
(start 500 100)
(control-left-right FACE 100 move-picture draw-losh)
```

最后一个表达式创建一个图形用户界面，允许用户通过使用方向键移动图形 *FACE*。用户每按一次方向键，图形移动 100 个像素（向右），或者-100 个像素（向左）。这个教学包还提供了其他方向上的移动控制按键。请使用它们来开发其他移动图片的程序。



第 11 章

自然数

到目前为止，我们所看到的惟一自引用数据是用 cons 构造的、任意长的表。之所以需要这样的数据定义，是因为要处理包含任意数目数据的表。自然数是一种可能含有任意多元素的数据类型；毕竟，自然数没有上界，至少理论上没有，以自然数为参数的函数应当能处理任意大的自然数。

在这一章中，我们将学习如何用自引用的数据定义来描述任意大的自然数，以及如何系统地开发处理自然数的函数。这样的函数有许多种类型，所以我们会学习几种不同的定义方法。

11.1 定义自然数

通常人们用枚举的方式引出自然数的定义：0，1，2，等等[1]。最后的“等等”表示这个序列就按照这样的方法继续下去。数学家和数学教师们通常使用“……”表示同样的含义。不过对我们来说，如果要系统地设计以自然数为参数的函数，无论是“等等”还是“……”都是不行的。所以，现在的问题是，“等等”到底是什么意思，或者换一种说法，自然数完整的、自引用的描述到底是什么？

唯一能把“等等”从描述自然数的枚举方法中去除的方法是使用自引用，第一种尝试是：

0 是自然数。

如果 n 是自然数，那么比 n 大 1 的数也是自然数。

虽然这种描述并不是十分严格[2]，但对于 Scheme 格式的数据定义来说，它是一个很好的开始：

> *natural-number*(自然数)是以下两者之一
> 1、0
> 2. (add1 *n*) 如果 *n* 是自然数。

操作 add1 把 1 加到一个自然数之上。当然，我们也可以使用(+ ...1)，但是，对于阅读数据定义和相关函数的人来说，add1 突出了“自然数”，而不是任意的数。

虽然对自然数我们已经非常地熟悉了，但是用这个数据定义来构造几个例子还是有意义的，显然，

```
0
```

是第一个自然数，所以

```
(add1 0)
```

是下一个。接着就很明显了：

```
(add1 (add1 0))
(add1 (add1 (add1 0)))
```

[1] 从 0 开始计数是非常重要的，因为这样，我们就可以用自然数表示某个表中元素的个数或者家谱树中的成员数。

[2] 要严格地定义自然数，必须使用集合论的知识

```
(add1 (add1 (add1 (add1 0))))
```

这些例子使我们想到了表的构造过程。我们从 empty 出发，通过用 cons 连接更多的元素，构造出表。现在，我们从 0 出发，通过（不断地）使用 add1 加上 1，构造出自然数。另外，可以使用经典的自然数缩写形式，如，(add1 0)的缩写是 1，(add1 (add1 0))的缩写是 2，等等。

处理自然数的函数必须能够提取构造自然数时所使用的数，就像处理表的函数能够提取出 cons 结构中的表一样。执行这种“提取”的操作被称为 sub1，它的运算规则是：

```
(sub1 (add1 n)) = n
```

它与 rest 操作的运算规则：

```
(rest (cons a-value a-list)) = a-list
```

相似。当然，我们也可以用(- *n* 1)，但是 sub1 突出了这个函数是作用于自然数之上。

11.2　处理任意大的自然数

下面设计函数 *hellos*，该函数以自然数 *n* 为参数，输出是由 *n* 个'hello 构成的表，该函数的合约为：

```
;; hellos : N -> list-of-symbols
;; 建立包含n个'hello的表
(define (hellos n) ...)
```

下面是一些例子：

```
(hellos 0)
;; 预期值:
empty

(hellos 2)
;; 预期值:
(cons 'hello (cons 'hello empty))
```

hellos 模板的设计遵循自引用数据定义的设计诀窍。显然 *hellos* 是个条件函数，其 cond 表达式包含两个子句，其中第一个子句必须区分 0 和其他可能的输入：

```
(define (hellos n)
  (cond
    [(zero? n) ...]
    [else ...]))
```

另外，数据定义显示，0 是原子数值，而其他的自然数是“包含”加 1 操作的复合值。这样，如果 n 不是 0，就从 n 中减去 1，得到的结果还是自然数。所以遵照设计诀窍，有：

```
(define (hellos n)
  (cond
    [(zero? n) ...]
    [else ... (hellos (sub1 n)) ... ]))
```

现在我们已经利用了数据定义给出的提示信息，下面开始定义函数。

假设(zero? *n*)的计算结果为真，正如例子所示，那么答案就应该是 empty。接着假设输入大于 0，不妨具体一点，我们令它为 2。按照模板的指示，*hellos* 应当要用到(*hellos* 1)来给出答案。函数的用途说明表明，(*hellos* 1)给出(cons 'hello empty)，它是仅包含一个'hello 的表。一般来说，(*hellos* (sub1 *n*))给出包

含 n－1 个'hello 的表。显然，要产生含有 *n* 个元素的表，必须用 cons 把'hello 和这个表连接起来：

```
(define (hellos n)
  (cond
    [(zero? n) empty]
    [else (cons 'hello (hellos (sub1 n)))]))
```

跟往常一样，最终的定义只是在模板的基础上添加少量的东西。

让我们手工计算来测试 *hellos*：

```
  (hellos 0)

= (cond
    [(zero? 0) empty]
    [else (cons 'hello (hellos (sub1 0)))])

= (cond
    [true empty]
    [else (cons 'hello (hellos (sub1 0)))])

= empty
```

这证明了 *hellos* 对第一种输入能够正常工作。接着考虑另一个例子：

```
  (hellos 1)

= (cond
    [(zero? 1) empty]
    [else (cons 'hello (hellos (sub1 1)))])

= (cond
    [false empty]
    [else (cons 'hello (hellos (sub1 1)))])

= (cons 'hello (hellos (sub1 1)))

= (cons 'hello (hellos 0))

= (cons 'hello empty)
```

最后一步利用已知的计算结果，(*hellos* 0)等于 empty，因此把带下划线的表达式替换成 empty。

最后证明函数对于下面的例子也能正常工作：

```
  (hellos 2)

= (cond
    [(zero? 2) empty]
    [else (cons 'hello (hellos (sub1 2)))])

= (cond
    [false empty]
```

```
      [else (cons 'hello (hellos (sub1 2)))])

  = (cons 'hello (hellos (sub1 2)))

  = (cons 'hello (hellos 1))

  = (cons 'hello (cons 'hello empty))
```

利用(*hellos* 1)的计算结果，我们极大地缩短了计算过程。

习题

习题 11.2.1 把 *hellos* 一般化为 *repeat*，该函数读入自然数 *n* 和一个符号，返回包含 *n* 个符号的表。

习题 11.2.2 设计函数 *tabulate-f*，把函数 *f* 应用于一些由自然数值组成的表，其中 *f* 是：

```
;; f : number -> number
(define (f x)
  (+ (* 3 (* x x))
     (+ (* -6 x)
        -1)))
```

具体地说，函数读入自然数 *n*，返回由 *n* 个 *posn* 结构体组成的表，表的第一个元素是点(*n*(*fn*))，第二个元素是点(*n-1*(*f n-1*))，以此类推。

习题 11.2.3 设计 *apply-n*，该函数读入自然数 *n*，把习题 10.3.6 中的 *move* 函数 *n* 次作用于习题 10.3.1 中的形状表 *FACE*，每一次作用都平移一个像素。这个函数的功能与 10.3 节有关，其目的是在画布上呈现连续移动的图形。

习题 11.2.4 表的成员也可以是表，即数据可以是表的表，甚至可以嵌套多层。下面就是这种思想下的一个极端的数据定义：

> *deep-list*（深层表）是下列两者之一：
> 1. *s*，其中 *s* 是符号。
> 2. (cons *dl* empty)，其中 *dl* 是深层表。

设计函数 *depth*，该函数读入一个深层表，测定这个表用了多少次 cons 来构成。设计函数 *make-deep*，该函数读入符号 *s* 和自然数 *n*，返回包含 *s*，使用 *n* 次 cons 构成的表。

11.3 补充练习：创建表，测试函数

在编程中，我们经常会使用数表。如，要用很大的表，而不是手工生成的小表来测试第 10.2 节中的函数 *extract1*，这样的表可以使用递归以及随机数发生器来生成。

习题

习题 11.3.1 Scheme 提供了操作 random，该操作读入一个比 1 大的自然数 *n*，返回一个在 *0* 和 *n-1* 之间的随机整数：

```
;; random : N -> N
```

```
;;给出一个在 0 和 n-1 之间的自然数
(define (random nm) ...)
```

连续两次调用(random *n*)所产生的结果可能是不一样的。

现在考虑如下的定义：

```
;; random-n-m : integer integer -> integer
;; ...
;; 假设: n < m
(define (random-n-m n m)
  (+ (random (- m n)) n))
```

给出 *random-n-m* 的简明而精确的用途说明。在数轴上用区间表示(random *n*)的返回值，通过计算证明你的解释。

习题 11.3.2　设计函数 *tie-dyed*，该函数读入一个自然数，返回一个数表，表中每个数在 20 和 120 之间。使用 *tie-dyed* 来实现习题 9.5.8 中的 *draw-circles*。

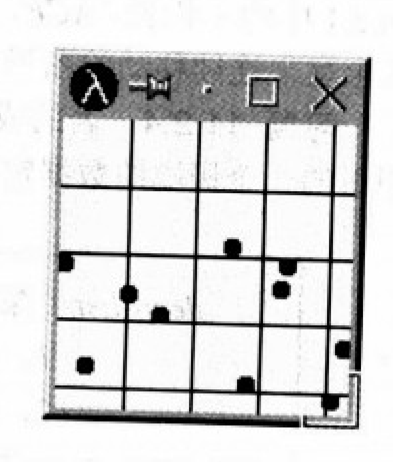
习题 11.3.3　设计函数 *create-temps*，该函数读入自然数 *n* 以及两个整数（分别称作 *low* 和 *high*），返回由 *n* 个 *low* 和 *high* 之间的整数构成的表。使用 *create-temps* 测试习题 9.5.4 中的 *check-range*。

最后，讨论如下问题：是否可以直接把 *create-temps* 的返回值传给 *check-range*，还是必须先知道 *create-temps* 生成的表是什么？有没有这样的 *low* 和 *high* 的值，我们无需知道 *create-temps* 的返回值，就可以预言测试的结果？用自动生成的数据进行函数测试，告诉了我们什么？

习题 11.3.4　开发函数 *create-prices*，该函数读入一个自然数，返回相应的价格表，表中的价格都在.10 元和 10.00 元之间，其中最小的计价单位是角。用 *create-prices* 来测试习题 9.5.3 中的 *dollar-store*?。

提示：在 10 元和 10.00 元之间，总共有多少金额是角的整数倍？

习题 11.3.5　设计一个函数，演示一次校园恶作剧。一小群学生聚集在一起，做了一些灌满颜料（只使用红颜料 *RED*）的塑料袋，夜间，带着塑料袋进入学校剧院，把所有的塑料袋都扔到座位上。程序唯一的输入应当是自然数，表示扔出的塑料袋的数量。恶作剧的再现由一块带黑格子的画布表示：



假设扔到剧院座位上的塑料袋的位置是随机分布的。格子中的每个方框代表一个座位。设计程序，使得改变其中一个参数的值导致格子的行数发生变化，而改变另一个参数的值导致格子的列数发生变化。

提示：开发一些辅助函数，能够在水平以及垂直方向上绘制出给定数量的直线。

11.4　自然数的另一种数据定义

使用上述标准的自然数数据定义可以很方便地开发出各种类型的处理数的函数。例如，考虑前 *n* 个自然数相乘的函数，即，读入自然数 *n*，把所有在 0（不包括）和 *n*（包括）之间的数乘起来，这个函数称作阶乘，数学上的符号是*!*，它的合约很容易给出：

```
;; ! : N -> N
;; 计算n · (n - 1) · ... · 2 · 1
(define (!n)...)
```

它读入一个自然数，并返回一个自然数。

要确定它的输入与输出关系就需要有点技巧了。当然，我们知道 1、2 和 3 的乘积是什么，所以：

```
(= (! 3)
   6)
```

与此类似，

```
(= (! 10)
   3628800)
```

真正的问题是，如何处理输入 0。按照非正式的任务描述，*!*被假定产生所有 0（不包括）到参数 *n*（包括）之间的所有数的乘积。既然 *n* 就是 0，这个要求就显得非常奇怪，因为在 0（不包括）和 0（包括）之间没有数。按照数学上的传统解决这个问题，令(*!* 0)为 1。

剩下的问题就很简单了。*!*的模板显然就是处理自然数的函数的模板：

```
(define (! n)
  (cond
    [(zero? n) ...]
    [else ... (! (sub1 n)) ...]))
```

第一个 cond 子句的答案已经有了，是 1。在第二个子句中，递归产生了前 $n-1$ 个数的乘积。要得出前 *n* 个数的乘积，只需用 *n* 去乘递归的值。图 11.1 给出了*!*完整的定义。

习题

习题 11.4.1　分别手工计算和使用 DrScheme 计算(*!* 2)的值。除此之外，用 10、100 和 1000 测试*!*。

注意：这些表达式的返回值是非常大的数，远超过许多程序设计语言本身的表示能力。

假设我们现在要设计函数 *product-from-20*，计算在 20（不包括）和 *n*（包括）之间所有数的乘积，这里的 *n* 是一个大于 20 的数。这时，我们有几种不同的选择方案。第一种方法，我们可以定义一个函数，计算(*!* *n*)和(*!* 20)，再用后者去除前者。简单的数学计算表明这种方法确实能产生 20（不包括）和 *n*（包括）之间所有数的乘积：

$$\frac{n\cdot(n-1)\cdot\ldots 21\cdot 20\ldots 1}{20\cdot\ldots 1}=n\cdot(n-1)\cdot\ldots\cdot 21\cdot\frac{20\cdot\ldots 1}{20\cdot\ldots 1}=n\cdot(n-1)\cdot\ldots\cdot 21$$

习题 11.4.2　使用这种方法定义函数 *product*，其参数是两个自然数，*n* 和 *m*，而且 $m > n$。它返回在 *n*（不包括）和 *m*（包括）之间所有数的乘积。

下面遵照设计诀窍，从精确描述函数的输入开始定义函数。显然，输入属于自然数，但是我们知道的不止这一点，输入属于这样的数集合：20，21，22，……。下面是这种集合的数据定义：

> *Natural number*(**自然数**)[>=**20**]是下列二者之一：
>
> 1. 20
> 2. （add1 *n*）　如果 *n* 是自然数[>=20]。
>
> 注意：在合约中，我们使用 N[>=20]，而不是"自然数[>=20]"。

使用新的数据定义，我们可以给出 *product-from-20* 的合约：

```
;; product-from-20: N [>= 20] -> N
;; 计算n · (n - 1) · ... · 21 · 1
(define (product-from-20 n-above-20) ...)
```

描述 *product-from-20* 输入输出关系的第一个例子是：

```
(= (product-from-20 21)
   21)
```

既然这个函数给出 20（不包括）和输入（包括）之间所有数的乘积，那么(*product-from-20* 21)必定得出 21，类似地，

```
(= (product-from-20 22)
   462)
```

理由同上。最后，按照数学上的习惯，我们有

```
(= (product-from-20 20)
   1)
```

product-from-20 的模板只是简单地修改了阶乘的模板（或者是处理任何自然数的函数的模板）：

```
(define (product-from-20 n-above-20)
  (cond
    [(= n-above-20 20) ...]
    [else ... (product-from-20 (sub1 n-above-20)) ...]))
```

其中输入 *n-above-20* 是 20 或者更大的数，如果它是 20，按照数据定义，它没有任何的组成成分。否则，它就是某个自然数[>=20]加 1 得到的结果，所以我们能把它减 1，恢复出它的“成分”。用这样的选择器表达式得出的值与输入属于同一种类型数据，所以应该使用递归。

要完成这个模板很简单。如上所述，(*product-from-20* 20)等于 1，这决定了第一个 cond 子句的答案。在其他情况下，(*product-from-20* (sub1 *n-above-20*))已经生成了在 20（不包括）和 *n-above-20*－1 之间所有数的乘积，唯一还没有放入这个范围的数就是 *n-above-20*。因此，(* *n-above-20* (*product-from-20* (sub1 *n-above-20*)))就是第二个子句的答案。图 11.1 是 *product-from-20* 完整的定义。

```
;; ! : N -> N
;; 计算 n · (n - 1) ·  ...  · 2 · 1
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n)))]))

;; product-from-20: N [>= 20] -> N
;; 计算 n · (n - 1) ·  ...  · 21 · 1
(define (product-from-20 n-above-20)
  (cond
    [(= n-above-20 20) 1]
    [else (* n-above-20 (product-from-20 (sub1 n-above-20)))]))

;; product: N[limit] N[>= limit] -> N
;; 计算 n · (n - 1) ·  ...  · (limit + 1) · 1
(define (product limit n)
  (cond
    [(= n limit) 1]
    [else (* n (product limit (sub1 n)))]))
```

图 11.1 计算阶乘、product-from-20 以及 product

习题

习题 11.4.3 设计 *product-from-minus-11*。这个函数读入一个大于或等于-11 的整数 *n*，返回在-11（不包括）和 *n*（包括）之间所有数的乘积。

习题 11.4.4 在习题 11.2.2 中，我们开发了一个函数，能够把某个函数 *f* 在区间（0，*n*）上的值列

成表格。设计函数 *tabulate-f20*，把某个函数 *f* 作用于大于 20 的自然数的值列成表格。具体来说，这个函数读入一个大于等于 20 的自然数 *n*，返回一个 *posn* 表，表中的每个元素都是结构体(make-posn *m* (*f* *m*))，其中 *m* 是 20（不包括）和 *n*（包括）之间所有的数。

比较*!*和 *product-from-20*，我们就能知道应如何设计把某个范围中所有的自然数连乘起来的函数。大致上，*product* 与 *product-from-20* 相似，除了限值 *limit* 不再是函数的一个部分，而是另一个输入，这说明其合约是：

```
;; product: N N -> N
;; 计算n · (n - 1) ·  ...  ·(limit + 1) · 1
(define (product limit n) ...)
```

product 的用途与 *product-from-20* 类似，计算从 *limit*（不包括）到某个大于 *limit* 的 *n*（包括）之间所有数的乘积。

不幸的是，对比 *product-from-20* 的合约，*product* 的合约相当不精确。具体来说，它并没有描述第二个参数所属的集合。根据它的第一个参数 *limit*，我们知道第二个参数属于"*limit*、(add1 *limit*)、(add1(add1 *limit*))，等等"。虽然我们能够轻易地列举出第二个参数可能的范围，但是这个返回取决于第一个参数，这是一种我们以前没有碰到过的、不寻常的情况。

尽管如此，如果我们假设 *limit* 是某个数，第二个参数的数据描述显然是：

> 假设 *limit* 是自然数，*natural number*(自然数)[>=*limit*](**N**[>=*limit*])是下列二者之一：
> 1. limit
> 2. （add1 *n*）如果 *n* 是自然数[>=*limit*]。

换句话说，这个定义类似于自然数[>= *limit*]的定义，只是把 20 替换成了变量 *limit*。当然，在中学，我们把自然数定义为 **N**[>=*0*]，把正整数定义为 **N**[>=*1*]。

有了这个新的数据定义，我们应该这样给出 *product* 的合约：

```
;; product: N[limit] N [>= limit] -> N
;; 计算n · (n - 1) ·  ...  · (limit + 1) · 1
(define (product limit n) ...)
```

更确切地说，我们用符号[*limit*]命名第一个参数（自然数），然后用这个名字来指定第二个参数。

剩下的程序开发工作就相当简单了，只要把 *product-from-20* 中的 20 全部替换成 *limit* 就基本可以了。唯一的改动是函数模板中的自然递归部分，因为现在函数需要两个参数，即 *limit* 和 N[>= *limit*]，所以模板中调用 *product* 时必须也使用两个参数，即 *limit* 和(sub1 *n*)：

```
(define (product limit n)
  (cond
    [(= n limit) ...]
    [else ... (product limit (sub1 n)) ...]))
```

图 11.1 给出了这个函数的完整定义。

习题

习题 11.4.5 在习题 11.2.2 和习题 11.4.4 中，我们开发了能把函数 *f* 在某些自然数上（分别是从某个数到 0 和从某个数[>= 20]到 20）的值列成表格的函数。开发函数 *tabulate-f-lim*，以类似的方式，把 *f* 从某个自然数 *n* 到另一个自然数 *limit* 的值列成表。

习题 11.4.6 在习题 11.2.2、习题 11.4.4 以及习题 11.4.5 中，我们开发了能够在不同的区间上列出

函数*f*的值的函数。这三个函数产生的都是降序排列的*posn*表。更具体地说，表达式(*tabulate-f* 3)产生下面这样的表：

```
(cons (make-posn 3 2.4)
  (cons (make-posn 2 3.4)
    (cons (make-posn 1 3.6)
      (cons (make-posn 0 3.0)
        empty))))
```

如果希望得到升序排列的*posn*表，我们必须使用另外一种数据集合，即小于（或等于）某个数的自然数：

> *natural numberl* 自然数[<=20]（N[<=20]是下列二者之一：
> 1. 20
> 2. (sub1 n) 如果 n 是自然数[<=20]。

当然，在中学里，我们把 N[<= -1]称为负整数。

设计函数：

```
;; tabulate-f-up-to-20 : N [<=20] -> N
(define (tabulate-f-up-to-20 n-below-20) ...)
```

该函数列出*f*关于小于 20 的自然数的值。具体来说，这个函数读入一个小于等于 20 的自然数，返回由*posn*结构体组成的表，表中的每个元素为(make-posn *m* (*f m*))，其中*m*是在 0 和*n*（包括）之间所有的数。

习题 11.4.7 设计函数 *is-not-divisible-by<=i*，该函数读入自然数 *i* [>=1]，以及自然数 *m*，而且 *i* < *m*。如果 *m* 不能被 1（不包括）和 *i*（包括）之间的任何一个数整除，函数就返回 true；否则，函数的返回值就是 false。使用 *is-not-divisible-by<=i* 来定义 *prime?*，该函数判断参数是不是素数。

11.5 更多与自然数有关的性质

自然数只是 Scheme 中数的一个很小的子集，因此上述的函数模板不能用来处理任意的数，特别是不能处理不精确的数。尽管如此，对于既能够处理自然数，又能处理其他的 Scheme 数的函数来说，这些模板是一个很好的开始。为了说明这一点，让我们来设计函数 *add-to-pi*，该函数读入自然数 *n*，产生 *n* +3.14，而且不使用+。

遵照设计诀窍，我们从下面的定义开始：

```
;; add-to-pi : N -> number
;; 不使用+，计算n + 3.14
(define (add-to-pi n) ...)
```

另一个容易的步骤是为一些示例输入确定输出：

```
(= (add-to-pi 0) 3.14)
(= (add-to-pi 2) 5.14)
(= (add-to-pi 6) 9.14)
```

函数 *hellos* 的合约（参见习题 11.2.1）和 *add-to-pi* 的合约之间的差别是输出不同，但是，正如我们所看到的，这并不影响模板的设计。适当使用 *hellos* 的模板，我们得到 *add-to-pi* 的模板如下：

```
(define (add-to-pi n)
  (cond
    [(zero? n) ...]
    [else ... (add-to-pi (sub1 n)) ... ])))
```

结合例子，这个模板让我们想到应该如何完成函数。如果输入是 0，*add-to-pi* 的输出是 3.14。在其他情况下，(*add-to-pi* (sub1 *n*))给出(- *n* 1) +3.14；既然正确的答案是这个值再加上 1，第二个 cond 子句的答案就应该是(add1 (*add-to-pi* (sub1 *n*)))。图 11.2 给出了完整的函数定义。

```
;; add-to-pi : N -> number
;; 不使用+，计算 n+3.14
(define (add-to-pi n)
  (cond
    [(zero? n) 3.14]
    [else (add1 (add-to-pi (sub1 n)))]))
```

图 11.2 把自然数加到 pi 之上

习题

习题 11.5.1 定义 *add*，该函数读入两个自然数 *n* 和 *x*，不使用 Scheme 提供的+，返回 *n*+*x*。

习题 11.5.2 设计函数 *multiply-by-pi*，该函数读入一个自然数，不使用 Scheme 提供的*，返回这个数乘上 3.14。例如：

```
(= (multiply-by-pi 0) 0)
(= (multiply-by-pi 2) 6.28)
(= (multiply-by-pi 3) 9.42)
```

定义函数 *multiply*，其输入是两个自然数 *n* 和 *x*，不使用 Scheme 提供的*，返回 *n* * *x*。最后，去除这些定义之中的+。

提示：*n* 乘以 *x* 就是把 *n* 个 *x* 加起来。

习题 11.5.3 设计函数 *exponent*，其输入是自然数 *n* 和数 *x*，计算

$$x^n.$$

最后，除去定义中的*。

提示：*x* 的 *n* 次方的意思是把 *n* 个 *x* 乘起来。

习题 11.5.4 深层表（参见习题 11.2.4）是另外一种表示自然数的方法。说明如何（用深层表来）表示 0、3 以及 8。设计函数 *addDL*，该函数读入两个深层表，分别代表两个自然数 *n* 和 *m*，输出代表 *n*+*m* 的深层表。

三论函数复合



第 3 章谈到程序是函数定义（包括可能的变量定义）的集合。为了指导函数设计，我们给出了一个大概的原则：

对于问题描述中的每一种依赖关系，定义一个辅助函数。

迄今为止，这个原则还是相当有效的，但是现在到了再一次考虑这个问题，给出另一个有关辅助函数设计原则的时候了。

本章第 1 节讨论辅助函数设计方针的改进，主要是把习题中得到的经验进行总结并形成文字，12.2 节和 12.3 节进行更深入的讨论，最后一节是补充练习。

12.1 设计复杂的程序

设计程序时，我们总是希望只用一个函数就可以实现目标，但往往需要使用辅助函数。特别地，如果问题描述涉及多种依赖关系，自然的方法是一个函数表示一种依赖关系，由此别人可以方便地阅读你的程序。第 3.1 节中有关电影院的例子就很好地说明了这种方法。

另外，按照设计诀窍，设计程序应该从严格分析输入和输出之间的关系开始。根据数据分析结果，先设计一个模板，然后进一步完善，最后得到完整的函数定义。从模板得到完整的函数定义意味着需要把模板中的表达式联系起来以组成问题的解答。这样做的时候，可能会遇到如下几种情况：

1. 如果问题的解答需要对某个变量的值进行分析，那么使用 cond 表达式；
2. 如果计算需要用到某个特定领域的知识，例如，绘图、会计、音乐等学科知识，那么使用辅助函数；
3. 如果某个计算必须处理表、自然数或是其他任意大的数据，那么使用辅助函数；
4. 如果函数的自然形态不是我们所期望的表达式，它很有可能就是程序目标的一般形式。在这种情况下，主函数是一个简短的定义，而计算由一般化的辅助函数完成。

后两种情况我们还没有讨论过，接下来的两节就使用两个例子对其进行说明。

当决定使用辅助函数时，应该把函数的合约、头部和用途说明加入到函数的清单中。

> **函数清单的原则**
>
> 维护一张函数清单，其中放置程序所需的函数。按照设计诀窍开发每一个函数。

把函数加入函数清单之前，先要检查是否已经存在相似的函数，或是清单中已经有了类似的函数说明。Scheme 提供了多种基本操作和函数，我们应该尽可能加以利用。

按照上述方针，我们逐一开发所需的函数，如果所设计的函数不依赖于清单中的其他函数，就可以进行测试，一旦完成了基本函数的测试，就可以测试其他的（调用基本函数的）函数，直到清单中的所有函数都测试完为止。我们应该在测试一个函数之前严格测试被它调用的其他函数，这样可以减少此后

逻辑错误的定位时间。

12.2　递归的辅助函数

人们总是需要对事物进行排序。咨询顾问按照回报多少对投资意向进行排序，医生对需要进行器官移植的患者进行排序，邮件程序对信件消息进行排序。一般来说，按照某种标准对一些值进行排序是许多程序必须完成的任务。

我们先来学习如何对数表进行排序，它是辅助函数设计的一个很好例子。排序函数读入一个表，产生另一个表。实际上，这两个表包含的数虽然相同，但是在输出中，数是按照某种顺序进行排列的。下面就是排序函数的合约和用途说明：

```
;; sort : list-of-numbers -> list-of-numbers
;; 使用表 alon 中的数，创建有序表
(define (sort alon) ...)
```

接着是两个例子：

```
(sort empty)
;; 预期值：
empty

(sort (cons 1297.04 (cons 20000.00 (cons -505.25 empty))))
;; 预期值：
(cons 20000.00 (cons 1297.04 (cons -505.25 empty)))
```

empty 不包含任何元素，可以认为是有序的，因此对应的输出是 empty。

下一步是把数据定义转变为函数模板。前面已经处理过数表，所以这一步很简单：

```
(define (sort alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (sort (rest alon)) ...]))
```

有了这个模板，下面就来处理程序最有意义的部分。从最简单的开始，我们分别考虑每一个 cond 子句，如例子所示，如果 *sort* 的输入是 empty，那么它的输出也是 empty。假设输入不是 empty，要处理的就是第 2 个 cond 子句，该子句含有两个表达式，按照设计诀窍，我们先必须弄明白它们的计算结果是什么：

1．(**first** *alon*)取出输入的第一个数；

2．按照函数用途说明，(*sort* (rest *alon*))是对(rest *alon*)进行排序的结果。

把这两个值组合在一起，意味着在合适的位置将第 1 个数插进第 2 个值，而后者是一个有序表。

我们来考虑第二个例子。如果 *sort* 的参数是(cons 1297.04 (cons 20000.00 (cons -505.25 empty)))，那么：

1．(first alon)是 1297.04；

2．(rest *alon*)是(cons 20000.00 (cons -505.25 empty))；

3．(*sort* (rest *alon*))返回(cons 20000.00 (cons -505.25 empty))。

要得到所需的结果，必须把 1297.04 插到表中某两个数之间，即第 2 个 cond 子句是一个表达式，它把(first *alon*) 插入到有序表(*sort* (rest *alon*))之中。

把一个数插入到一个有序表之中并不是一件简单的任务。在得知合适的插入位置之前，必须从头到

尾遍历整个表。不过，表的遍历可以单独使用一个函数。因为表的大小是任意的，而处理任意大小的值需要递归函数，因此，我们必须设计一个递归的辅助函数，该辅助函数以一个数和一个有序表为参数，结果为包含两者的有序表，假定该函数的名字为 *insert*，则：

```
;; insert:number list-of-numbers -> list-of-nrmbers
;; 用表 alon 中的数和 n，创建降序排列的表；
;; alon 已被排好序
(define (insert n alon) ...)
```

基于 *insert*，我们可以很方便地给出 *sort* 的定义：

```
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (sort (rest alon)))]))
```

第二个子句的意思是，要产生最终的值，可先取出非空表的第一个元素，再对其余部分进行排序，最后使用 *insert* 把前者插入到后者的合适位置上。

下面开始设计 *insert*，函数合约、头部及用途说明已经有了，现在需要的是构造一些例子。既然 *insert* 的第一个输入是原子值，我们就以表的数据定义为基础来构造例子。即，先考虑如果参数是一个数和 empty，*insert* 应该干什么。按照 *insert* 的用途说明，输出应该是一个表，该表除了第一个参数外还必须包含第二个参数中的所有数据，因此：

```
(insert 5 empty)
;; 预期值：
(cons 5 empty)
```

除了 5 以外，我也可以使用其他任何数。

第二个例子必须使用一个非空表，而此时，前面所讨论的 *sort* 是如何处理非空表的例子就很能说明 *insert* 的思想，具体来说，*sort* 必须把 1297.04 插入到(cons 20000.00 (cons -505.25 empty))中合适的位置：

```
(insert 1297.04 (cons 20000.00 (cons -505.25 empty)))
;; 预期值：
(cons 20000.00 (cons 1297.04 (cons -505.25 empty)))
```

与 *sort* 不同，*insert* 使用了 2 个参数，第 1 个参数是一个数，是原子值，所以应该把注意力集中到第 2 个参数，即数表上，这表明我们又一次要用到处理表的模板：

```
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (insert n (rest alon)) ...]))
```

该模板和 *sort* 模板之间的唯一区别是它还需要考虑另外一个参数 *n*。

要填写 *insert* 模板中的空白部分，需要考虑不同的情况。第一种情况是空表，按照用途说明，此时 *insert* 必须构造一个只含有元素 *n* 的表，所以第一种情况下的答案就是(cons *n* empty)。

第二种情况就比较复杂了，如果 *alon* 不为空，则：

1. (first *alon*)是 *alon* 的第一个数。
2. (*insert n* (rest *alon*))产生一个有序表，该表包含 *n* 和(rest *alon*)中的所有数据。

现在的问题是，如何把这些数据结合起来，组成所需的答案。让我们考虑如下的例子：

```
(insert 7 (cons 6 (cons 5 (cons 4 empty))))
```

这里 7 比第二个输入中的所有数都大，所以应该用 cons 把 7 和(cons 6 (cons 5 (cons 4 empty)))结合在

一起。反之，如果是：

```
(insert 3 (cons 6 (cons 2 (cons 1 (cons -1 empty)))))
```

n 就必须被插入到表的其余部分之中，更具体地说：

1．(first *alon*)是 6。

2．(*insert n* (rest *alon*))是(cons 3 (cons 2 (cons 1 (cons -1 empty))))。

使用 cons 把 6 加入到表中，就得到了所需的答案。

现在，考虑这些例子的一般特性，这需要再一次区分不同的情况。如果 *n* 比(first *alon*)大（或是一样大），因为 *alon* 是一个有序表，*alon* 中的所有数都不会大于 *n*。如果 *n* 比(first *alon*)小，那么就是还没有找到插入 *n* 的合适位置。这时第一个元素必定是(first *alon*)，而 *n* 必须被插入(rest *alon*)之中。在这种情况下，最终的结果是：

```
(cons (first alon) (insert n (rest alon)))
```

该表包含了 *n* 和 *alon* 的所有元素，这正是我们所需的。

把这些讨论翻译成条件表达式，结果为：

```
(cond
  [(>= n (first alon)) ...]
  [(<  n (first alon)) ...])
```

接着，只需填入合适的表达式即可。图 12.1 给出了 *insert* 和 *sort* 的完整定义。

术语：这种排序方法在程序设计书籍中，称作插入排序。

```
;; sort : list-of-numbers -> list-of-numbers (sorted)
;; 使用表 alon 的所有元素，创建有序数表。alon 被降序排序
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon) (sort (rest alon)))]))

;; insert : number list-of-numbers (sorted) -> list-of-numbers (sorted)
;; 使用 n 和表 alon 中的元素，创建降序排列的数表；
;; alon 是有序表
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
            [(>= n (first alon)) (cons n alon)]
            [(<  n (first alon)) (cons (first alon) (insert n (rest alon)))])]))
```

图 12.1 对数表排序

习题

习题 12.2.1 设计一个程序，按照日期对邮件进行排序。邮件结构体的定义如下：

```
(define-struct mail (from date message))
```

> *mail-message* 是结构体：
>
> (make-mail name n s)
>
> 其中 *name* 是字符串，*n* 是数，*s* 也是字符串。

另外再开发一个程序，按照名称对邮件进行排序。请使用 string<?比较两个字符串的大小。

习题 12.2.2 函数 *search*：

```
;; search : number list-of-numbers -> boolean
(define (search n alon)
  (cond
    [(empty? alon) false]
    [else (or (= (first alon) n) (search n (rest alon)))]))
```

判断一个数是否在一个表中出现。如果表不含该数，函数需要遍历整个表才能得出结论。利用表的有序性，开发函数 *search-sorted*，判断某个数是否在一个有序表中出现。

术语：在程序设计书籍中，称函数 *search-sorted* 实施的查找为线性查找。

12.3 问题泛化与函数泛化

考虑多边形的绘制。多边形是一种几何形状，有任意数量的顶点。一种自然的表示多边形的方法是使用由 *posn* 结构体组成的表：

> *posn* 表 *list-of-posas* 是下列二者之一：
> 1. 空白表 ***empty***，
> 2. (cons *p lop*)，其中 *p* 是 *posn* 结构体而 *lop* 是 *posn* 表。

每个 *posn* 代表多边形的一个顶点，例如：

```
(cons (make-posn 10 10)
  (cons (make-posn 60 60)
    (cons (make-posn 10 60)
      empty)))
```

表示了一个三角形。现在的问题是 empty 代表什么多边形。答案是 empty 并不代表多边形，所以 empty 的类型并不是多边形。多边形至少应有一个顶点，即表示多边形的表至少要包含一个 *posn* 结构体，因此，我们有以下数据定义

> polygon（多边形）是下列两者之一：
> 1. (cons *p* empty)，其中 *p* 是 *posn*。
> 2. (cons *p lop*)，其中 *p* 是 *posn*，而 *lop* 是多边形。

简而言之，使用由 *posn* 组成的表来表示多边形是不恰当的，数据定义的修改使我们更接近于目标，编程更简单。

因为绘图操作的返回值总是 true，自然我们可以得出如下的合约及用途说明：

```
;; draw-polygon : polygon -> true
;; 绘制 a-poly 所指定的多边形
(define (draw-polygon a-poly) ...)
```

换句话说，这个函数画出顶点之间的连线，在所有的连线都画出之后，函数返回 true。例如，应用函数于上面提到的 *posn* 表，得到的应该是一个三角形。

虽然现在的数据定义与常用的表的数据定义并不相同，模板仍然和表处理函数的模板类似：

```
;; draw-polygon : polygon -> true
;; 绘制 a-poly 所指定的多边形
(define (draw-polygon a-poly)
  (cond
    [(empty? (rest a-poly)) ... (first a-poly) ...]
    [else ... (first a-poly) ...
        ... (second a-poly) ...
        ... (draw-polygon (rest a-poly)) ...]))
```

由于数据定义的两种情况都用到了 cons，因此必须检查表的其余部分，在第一种情况下，它是 empty，在第二种情况下，它是非空的表。另外，在第一个子句中，可以加入(first *a-poly*)；而在第二个子句中，除了放入第一个元素外，还要放入第二个元素，毕竟，按照多边形定义，此时它至少包含两个 *posn*。

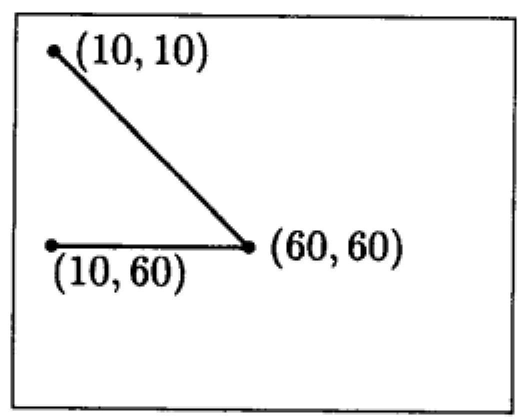
现在可以填写模板中的“……”以得到完整的函数定义。对于第一个子句，答案必定是 true，因为并不存在两个 *posn* 可以用于绘制一条连线。对于第二个子句，我们有两个 posn，可以在它们之间画一条连线，并且(*draw-polygon* (rest *a-poly*))能绘制出所有其余的连线。换句话说，我们可以在第二个子句中写上：

```
(draw-solid-line (first a-poly) (second a-poly))
```

如果一切正常，那么(*draw-solid-line* ...)和(*draw-poly* ...)都会返回 true，使用 and 连接这两个表达式，*draw-polygon* 就可以画出所有的直线，函数的定义如下：

```
(define (draw-polygon a-poly)
  (cond
    [(empty? (rest a-poly)) true]
    [else (and (draw-solid-line (first a-poly) (second a-poly))
             (draw-polygon (rest a-poly)))]))
```

不幸的是，若用上面提到的三角形对函数进行测试，我们可立即发现，这个函数并没有绘制出一个含有三条边的多边形，而是绘制出一条连接所有顶点的开放曲线：



用数学的语言说，我们得到的函数比所需要的更一般。我们刚才定义的函数应该叫做 connect-the-dots，而不是 *draw-polygon*。

要从这个一般的函数得到所需的函数，必须把最后一个点和第一个点连接起来。有几种方法可以做到这一点，而所有方法都要用到我们刚才定义的函数。换句话说，我们在一般的函数之上，再定义一个辅助函数。

一种方法是，定义一个新函数，把多边形的第一个顶点加到表的末端，然后用新生成的表来绘制图形；另一种方法是，把最后一个顶点添加到表的前端；还有一种方法是修改 *draw-polygon* 函数，使它能够把最后一个顶点和第一个顶点连接起来。这里讨论第二种方法，其他两种方法作为习题。

要把 *a-poly* 的最后一个元素加入到它的前端，我们需要：

```
(cons (last a-poly) a-poly)
```

其中 *last* 是个辅助函数，它的作用是提取非空表的最后一元素。事实上，定义了 *last*，就得到了 *draw-polygon* 的定义，参见图 12.2。

```
;; draw-polygon : polygon -> true
;; 绘制 a-poly 所指定的多边形
(define (draw-polygon a-poly)
  (connect-dots (cons (last a-poly) a-poly)))

;; connect-dots : polygon -> true
;; 绘制 a-poly 各点之间的连线
(define (connect-dots a-poly)
  (cond
    [(empty? (rest a-poly)) true]
    [else (and (draw-solid-line (first a-poly) (second a-poly),red)
               (connect-dots (rest a-poly)))]))

;; last : polygon -> posn
;; 提取 a-poly 的最后一个 posn
(define (last a-poly)
  (cond
    [(empty? (rest a-poly)) (first a-poly)]
    [else (last (rest a-poly))]))
```

图 12.2 绘制多边形

在函数清单中可以添加如下内容：

```
;; last : polygon -> posn
;; 提取 a-poly 中的最后一个 posn
(define (last a-poly) ...)
```

由于 last 的参数是多边形，还可以使用上面设计的模板：

```
(define (last a-poly)
  (cond
    [(empty? (rest a-poly)) ... (first a-poly) ...]
    [else ... (first a-poly) ...
        ... (second a-poly) ...
        ... (last (rest a-poly)) ...]))
```

把模板转变为完整的函数非常容易。如果表中只有一个元素，该元素就是所求的答案。如果(rest *a-poly*)非空，(*last* (rest *a-poly*))就会给出 *a-poly* 的最后一个元素。图 12.2 的最后部分是 *last* 的完整定义。

总的说来，*draw-polygon* 的设计使我们想到了一个更一般的问题：连接表中各点。通过定义辅助函数，使用更一般的函数（泛化函数），我们解决了问题。正如所看到的和将要看到的那样，泛化函数的使用是简化问题的最好方法。

习题

习题 12.3.1 设计辅助函数 *add-at-end*，其将表的第一个元素加到表的末端，接着修改函数 *draw-polygon*。

习题 12.3.2 修改 *connect-dots*，它的输入还包括一个 *posn* 结构体，表示与最后一个 *posn* 相连接

的点。请使用新的 *connect-dots* 函数，修改函数 *draw-polygon*。

累积器：这个新的 *connect-dots* 是累积器函数的一个实例。本书的第六部分将完整地讨论这类问题。

12.4 补充练习：字母的重新排列

报纸上经常会出现这样的游戏：用某些字母拼出所有可能的单词。玩这类游戏的一种方法是，系统列出这些字母所有可能的排列，然后检查其中哪些在字典中出现。假设给出的字母是“a”、“d”、“e”、“r”，它们共有二十四种排列：

ader	eadr	erad	drea	Ared
daer	edar	erda	arde	Raed
dear	edra	adre	rade	Read
dera	aerd	dare	rdae	Reda
aedr	eard	drae	rdea	

其中真正的单词有三个：“read”、“dear”和“dare”。

显然，系统列出所有可能的排列是计算机程序的任务。程序读入一个单词，返回对字母重新排序产生的单词表。

一种表示单词的方式是使用符号表。表中的每一个元素代表一个字母：'a,'b,…,'z。下面是单词的数据定义：

> word（单词）是下列两者之一：
> 1. `empty`，
> 2. (cons *a w*)，其中 *a* 是符号（`'a`，`'b`，……，`'z`）而 *w* 是单词。

习题

习题 12.4.1 给出单词表的数据定义，并系统构造单词的例子以及单词表的例子。

我们把这个函数函数称为 *arrangements*[1]，它的模板是一个表处理函数：

```
;; arrangements : word -> list-of-words
;;创建 a-word 中字母的所有排列
(define (arrangements a-word)
  (cond
    [(empty? a-word) ...]
    [else ... (first a-word) ... (arrangements (rest a-word)) ...]))
```

有了合约、数据定义和例子，现在可以观察模板中的每一个 cond 子句：

1．如果输入是 empty，重新排列只能得到一个结果：empty。所以，返回值是(cons empty empty)，即只含一个空表的表。

[1] 数学术语是：“排列”。

2. 否则，输入的单词至少包含一个字母，其中(first *a-word*)是第一个字母，而递归产生了其余所有字母的排列。例如，如果输入表是：

```
(cons 'd (cons 'e (cons 'r empty)))
```

那么(*arrangements*(cons 'e(cons 'r empty)))的结果是：

```
(cons (cons 'e (cons 'r empty))
  (cons (cons 'r (cons 'e empty))
    empty))
```

要获得所有可能的排列，必须把第一个元素（例子中的'd）插入到所有部分单词的所有可能位置（包括第一个位置和最后一个位置）。

把字母插入到单词的时候需要处理任意长的表，所以定义 *arrangements* 需要另一个函数（称为 *insert-everywhere/in-all-words*）：

```
(define (arrangements a-word)
  (cond
    [(empty? a-word) (cons empty empty)]
    [else (insert-everywhere/in-all-words (first a-word)
            (arrangements (rest a-word)))]))
```

习题 12.4.2 设计函数 *insert-everywhere/in-all-words*，该函数读入一个符号和一个单词表，它的返回值也是一个单词表，其中第一个参数被插入到第二个参数所包含的单词（从头到尾）的所有位置。

提示：再一次考虑上面提到的例子。若要把'd 插入到单词(cons 'e (cons 'r empty))和(cons 'r (cons 'e empty))之中，例子为：

```
(insert-everywhere/in-all-words 'd
  (cons (cons 'e (cons 'r empty))
    (cons (cons 'r (cons 'e empty))
      empty)))
```

记住，第二个参数对应于的单词序列为“er”和“re”。

可使用 Scheme 提供的操作 append，该函数读入两个表，结果是两个表的连接。如：

```
  (append (list 'a 'b 'c) (list 'd 'e))
= (list 'a 'b 'c 'd 'e)
```

我们会在第 17 章讨论诸如 append 这样的函数的设计方法。

用 list 构造表

第 13 章

使用 *cons* 构造一个包含多个元素的表十分麻烦，因此 Scheme 提供了 *list* 操作，该操作接受任意数量的值作为输入以创建一个表。下面是扩展的 Scheme 语法：

```
<prm>=list
```

扩展的 Scheme 值的集合是：

```
<val> = (list <val> ... <val>)
```

理解 *list* 表达式的一种简单方法是将它当作若干 cons 的简写，具体来说，就是将每个形如

```
(list exp-1 ... exp-n)
```

的表达式看成是如下表达式的简写：

```
(cons exp-1 (cons ... (cons exp-n empty)))
```

下面是三个例子：

```
(list 1 2)
= (cons 1 (cons 2 empty))

(list 'Houston 'Dallas 'SanAntonio)
= (cons 'Houston (cons 'Dallas (cons 'SanAntonio empty)))

(list false true false false)
= (cons false (cons true (cons false (cons false empty))))
```

它们分别产生包含 2 个、3 个和 4 个元素的表。

list 不仅可以作用于值，也可以作用于表达式：

```
(list (+ 0 1) (+ 1 1))
= (list 1 2)
```

在创建表之前，Scheme 先计算表达式，如果表达式计算产生错误，表就不会被创建：

```
(list (/ 1 0) (+ 1 1))
= /: divide by zero
```

简而言之，list 的行为和 Scheme 的其他基本操作完全一样。

使用 list 可以极大地简化表的表示，对于包含多个元素的表以及含有结构体的表特别有用。下面是一个例子：

```
(list 0 1 2 3 4 5 6 7 8 9)
```

该表包含了 10 个元素，如果使用 cons 和 empty，需要 10 个 cons 和 1 个 empty。类似地，下面的表：

```
(list (list 'bob 0 'a)
      (list 'carl 1 'a)
   (list 'dana 2 'b)
```

```
(list 'erik 3 'c)
(list 'frank 4 'a)
(list 'grant 5 'b)
(list 'hank 6 'c)
(list 'ian 8 'a)
(list 'john 7 'd)
(list 'karel 9 'e))
```

用了 11 个 list，而原先需要 40 个 cons 和 11 个 empty。

习题

习题 13.1.1 使用 cons 和 empty 表示下列表：

```
1. (list 0 1 2 3 4 5)
2. (list (list 'adam 0) (list 'eve 1) (list 'louisXIV 2))
3. (list 1 (list 1 2) (list 1 2 3))
```

习题 13.1.2 先确定每个表以及每个嵌套的表包含多少个元素，然后使用 list 表示下列表：

```
1. (cons 'a (cons 'b (cons 'c (cons 'd (cons 'e empty)))))
2. (cons (cons 1 (cons 2 empty)) empty)
3. (cons 'a (cons (cons 1 empty) (cons false empty)))
4. (cons (cons 1 (cons 2 empty)) (cons (cons 2 (cons 3 empty)) empty))
```

习题 13.1.3 在一些特殊的情况下，我们形成表时会同时使用 cons 和 list：

```
1. (cons 'a (list 0 false))
2. (list (cons 1 (cons 13 empty)))
3. (list empty empty (cons 1 empty))
4. (cons 'a (cons (list 1) (list false empty)))
```

改写它们，使其只包含 list。

习题 13.1.4 给出下列表达式的值：

```
1. (list (symbol=? 'a 'b) (symbol=? 'c 'c) false)
2. (list (+ 10 20) (* 10 20) (/ 10 20))
3. (list 'dana 'jane 'mary 'laura)
```

习题 13.1.5 给出下列表达式的值：

```
1. (first (list 1 2 3))
2. (rest (list 1 2 3))
```

使用 list 可极大简化了包含表的表达式计算，下面是 9.5 节中相关例子的递归计算步骤：

```
  (sum (list (make-ir 'robot 22.05) (make-ir 'doll 17.95)))
= (+ (ir-price (first (list (make-ir 'robot 22.05) (make-ir 'doll 17.95))))
     (sum (rest (list (make-ir 'robot 22.05) (make-ir 'doll 17.95)))))
= (+ (ir-price (make-ir 'robot 22.05))
     (sum (list (make-ir 'doll 17.95))))
```

这里第一次使用了与新的基本操作相关的等式：

```
= (+ 22.05
```

```
    (sum (list (make-ir 'doll 17.95))))
= (+ 22.05
    (+ (ir-price (first (list (make-ir 'doll 17.95))))
       (sum (rest (list (make-ir 'doll 17.95))))))
= (+ 22.05
    (+ (ir-price (make-ir 'doll 17.95))
       (sum empty)))
= (+ 22.05 (+ 17.95 (sum empty)))
= (+ 22.05 (+ 17.95 0))
```

对于list类型的值来说，first和rest运算规则的使用相当自然，因此不需要将list转化成cons和empty。

按照一种旧的程序设计语言的约定[1]，我们可以进一步简化表和符号的记法。如果一个表用 list 表示，按约定，可以简单地把 list 去除，即可以认为剩下的开括号就代表了开括号自身以及关键字 list。例如：

```
'(1 2 3)
```

表示

```
(list 1 2 3)
```

或者

```
(cons 1 (cons 2 (cons 3 empty)))。
```

类似地，

```
'((1 2) (3 4) (5 6))
```

代表了

```
(list (list 1 2) (list 3 4) (list 5 6)),
```

它们又可以进一步表示为含有 cons 和 empty 的表达式。

如果去除符号前面的引号，给出由符号组成的表就是一件轻而易举的事：

```
'(a b c)
```

这个简写表示了：

```
(list 'a 'b 'c)
```

更有意思的是，

```
'(<html>
   (<title> My First Web Page)
   (<body> Oh!))
```

代表了：

```
(list '<html>
  (list '<title> 'My 'First 'Web 'Page)
  (list '<body> 'Oh!))。
```

习题

习题 13.1.6 在必要的地方恢复 list 以及引号的使用：

1.

```
'(1 a 2 b 3 c)
```

2.

```
'((alan 1000)
```

[1] 该约定起源于 1958 年出现的高级程序设计语言 LISP。虽然 Scheme 是一种不同的语言，但它的诸多思想来自 *LISP*。

```
  (barb 2000)
  (carl 1500)
  (dawn 2300))
```

3.

```
'((My First Paper)
  (Sean Fisler)
  (Section 1
   (Subsection 1 Life is difficult)
   (Subsection 2 But learning things makes it interesting))
  (Section 2
   Conclusion? What conclusion?))
```



第三部分

再论任意大数据的处理





再论自引用数据定义

表和自然数是两种需要使用自引用数据定义描述的数据类型。它们的数据定义都由两个子句组成，其中都有一句是自引用的。不过，还有许多有意义的数据类型，它们需要比这更复杂的定义。事实上，数据定义的变化是没有止境的。因此，有必要学习如何由非正式的信息描述得出数据定义。一旦有了非正式的信息描述，遵循经过少许修改的设计诀窍，就可以给出自引用的数据定义。

14.1 结构体中的结构体

医学研究者使用家谱树来研究遗传疾病。例如，他们可能会在家谱树中查找某种特定的眼睛颜色。计算机可以帮助他们完成这些任务，所以很自然地，我们要考虑家谱树的表示法，并设计处理家谱树的函数。

记录某个家族家谱树的一种方法是，每当家族中有孩子出生时，向树中添加一个节点，在节点中给出到达它的父亲节点和母亲节点的连接，这些连接告诉我们在家谱树中人与人之间的关系是什么。对于不知道其父母是谁的人，就不必给出连接。这样得到的家谱树被称为祖先家谱树，因为给定树中任意一个节点，沿着箭头就可以找出这个人的祖先，但是不能找到他的后代。

在记录家谱树的同时，可能还要记录其他一些信息，如每个人的出生日期、出生时的体重、眼睛的颜色以及头发的颜色等。

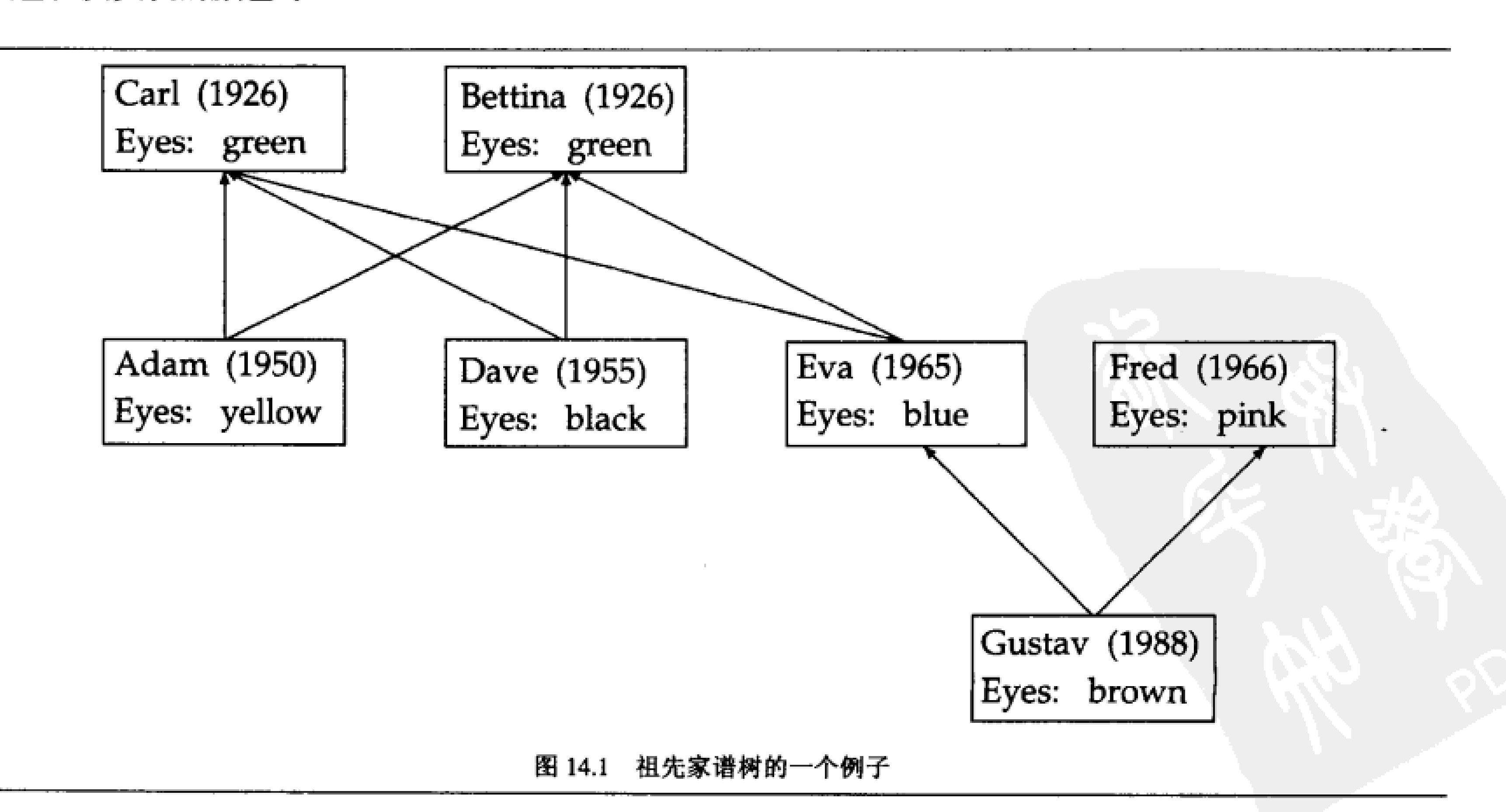


图 14.1 祖先家谱树的一个例子

祖先家谱树的图形表示请参见图 14.1。图中，Adam 是 Bettina 和 Carl 的孩子，他的眼睛是黄色的，出生于 1950 年。类似地，Gustav 是 Eva 和 Fred 的孩子，他的眼睛是棕色的，出生于 1988 年。要在家谱树中表示一个孩子，只需把这样几条信息结合起来：父亲的信息、母亲的信息、名字、出生日期以及眼睛的颜色。这表明我们需要一个新的结构体：

```
(define-struct child (father mother name date eyes))
```

child 结构体中的五个字段分别记录了所需的信息，相应的数据定义是：

> *child* 是结构体：
>
> **(make-child** *f m na da ec*)
>
> 其中 *f* 和 *m* 是 *child* 结构体；*na* 和 *ec* 是符号；*da* 是数。

虽然这个数据定义非常简单，但很不幸的是，它没什么用处。该定义引用了它自己，因为它没有任何子句，所以无法建立一个 *child* 结构体。如果试图建立一个 *child* 结构体，就不得不永无止境地写下去，如：

```
(make-child
  (make-child
    (make-child
      (make-child
        ...
        )))
  ... ... ... ...)
```

正是因为这个原因，所以我们（暂时）要求所有自引用的数据定义都包含多个子句，而且其中至少要有一个子句不引用数据定义自身。

我们暂时推迟讨论数据定义，转而研究怎样使用 *child* 结构体来表示家谱树。假设要在一棵现有的家谱树中添加一个孩子，而且已经有了其父母的数据表示。那么，我们可以建立一个新的 *child* 结构体。例如，对于 Adam 来说，假设 *Carl* 和 *Bettina* 代表了 Adam 的父母，我们可以建立如下的 *child* 结构体：

```
(make-child Carl Bettina 'Adam 1950 'yellow)
```

问题是，我们并不总是知道某个人的父母是谁，例如在图 14.1 所描述的家族中，我们就不知道 Bettina 的父母是谁。然而，即使不知道某人的父亲或母亲是谁，也必须使用某个 Scheme 值填入 *child* 结构体的两个（对应）字段。可以使用任意一种值来表示缺乏信息（5，false 或者'none）；这里，我们使用 empty。例如，要构造 Bettina 的 *child* 结构体，可以这样做：

```
(make-child empty empty 'Bettina 1926 'green)
```

当然，如果只是不知道某人父母中的一个，我们只需在相应的字段中填入 empty。

分析表明，*child* 节点有着如下的数据定义：

> *child* 节点是(**make-child** *f m na da ec*)，其中
>
> 1. *f* 和 *m* 是：
>   a. **empty** 或者
>   b. *child* 节点；
> 2. *na* 和 *ec* 是符号；
> 3. *da* 是数。

这个定义在两个方面很特别。第一，它是关于结构体的自引用数据定义；第二，这个数据定义在它的第一个成分和第二个成分中提到了两种选择。这违反了传统的数据定义形式。

定义另一种家谱树节点的集合，可以避免这个问题：

> *family-tree-node*（家谱树节点，简称 *ftn*）是下列两者之一：
> 1. **empty**。
> 2. (make-child *f m na da ec*)
>
> 其中 *f* 和 *m* 是 *ftn*，*na* 和 *ec* 是符号，*da* 是数。

这个新的定义符合我们的传统。它由两个子句组成，其中的一个子句是自引用的，另一个子句则不是。

与前一个结构体的数据定义相比，*ftn* 的定义并不是简单地解释哪个字段可以包含哪种数据类型，而是由多个子句组成的，是自引用的。考虑到这是我们遇到的第一个此种形式的数据定义，我们将图 14.1 中的例子仔细地转化成数据，从而确保这种新的数据类型可以表示所关心的信息。

把 Carl 的信息转化成一个 *ftn* 很容易：

```
(make-child empty empty 'Carl 1926 'green)
```

Bettina 和 Fred 可以用类似的节点来表示。相应地，Adam 的节点是这样建立的：

```
(make-child (make-child empty empty 'Carl 1926 'green)
            (make-child empty empty 'Bettina 1926 'green)
            'Adam
            1950
            'yellow)
```

正如例子所示，用简单的方法逐个转化节点数据会使用许多重复的数据。例如，如果像构建 Adam 的 *child* 结构体一样构建 Dave 的 *child* 结构体，就会得到：

```
(make-child (make-child empty empty 'Carl 1926 'green)
            (make-child empty empty 'Bettina 1926 'green)
            'Dave
            1955
            'black)
```

因此，一种较好的方法是，为每一个节点引入一个变量定义，而此后就使用变量。为了简单起见，我们用 *Carl* 来代表描述 *Carl* 的 *child* 结构体，以此类推。图 14.2 给出了把家谱树完整地转化成 Scheme 的结果。

```
;; 老一代人：
(define Carl (make-child empty empty 'Carl 1926 'green))
(define Bettina (make-child empty empty 'Bettina 1926 'green))

;; 中间一代人：
(define Adam (make-child Carl Bettina 'Adam 1950 'yellow))
(define Dave (make-child Carl Bettina 'Dave 1955 'black))
(define Eva (make-child Carl Bettina 'Eva 1965 'blue))
(define Fred (make-child empty empty 'Fred 1966 'pink))

;; 年轻一代人：
(define Gustav (make-child Fred Eva 'Gustav 1988 'brown))
```

图 14.2　家谱树例子的 Scheme 表示

图 14.2 中的结构体定义自然对应于多层嵌套的方框，每个方框都包含五个部分，前两个部分又各包含一个方框，而后者的两个部分中又包含方框，以此类推。因此，如果使用嵌套的方框来画出家谱树的结构体定义，我们很快就会被该图片的细节所淹没。另外，这样的图片会多次复制树的某个部分，如同

在不使用变量定义时试图使用 make-child 一样。因此，最好把结构体想象成方框和箭头，如同图 14.1。一般来说，程序员必须要能在这两种图形表示法之间灵活地切换。对于在结构体中提取值来说，“方框中的方框”更为适用；对于在相互连接的大量结构体的集合中寻找关系来说，“方框和箭头”更为适用。

有了对家谱树表示法的深刻理解，我们可以转而设计读入家谱树的函数。先来观察一般化的该类型函数：

```
;; fun-for-ftn : ftn -> ???
(define (fun-for-ftn a-ftree) ...)
```

毕竟，在建立模板时无需考虑函数的用途。

既然 *ftn* 的数据定义包含两个子句，那么模板也必须由包含两个子句的 cond 表达式组成。第一个子句处理 empty，第二个子句处理 *child* 结构体：

```
;; fun-for-ftn : ftn -> ???
(define (fun-for-ftn a-ftree)
  (cond
    [(empty? a-ftree) ...]
    [else ; (child? a-ftree)
      ... ]))
```

另外，对第一个子句来说，输入是原子的，所以我们没有其他的事情可以做。不过，对第二个子句来说，输入包含了五条信息，即另外两个家谱树节点、人名、出生日期以及眼睛的颜色：

```
;; fun-for-ftn : ftn -> ???
(define (fun-for-ftn a-ftree)
  (cond
    [(empty? a-ftree) ...]
    [else
      ... (fun-for-ftn (child-father a-ftree)) ...
      ... (fun-for-ftn (child-mother a-ftree)) ...
      ... (child-name a-ftree) ...
      ... (child-date a-ftree) ...
      ... (child-eyes a-ftree) ...]))
```

因为数据定义的第二个子句是自引用的，所以我们还把 *fun-for-ftn* 作用于 *father* 和 *mother* 字段。

现在来处理一个具体的例子：*blue-eyed-ancestor?*，该函数判断某个给定的家谱树中有没有人的眼睛是蓝色的：

```
;; blue-eyed-ancestor? : ftn -> boolean
;; 判断 a-ftree 是否包含一个 child 结构体，
;; 其 eyes 字段为 'blue
(define (blue-eyed-ancestor? a-ftree) ...)
```

遵照诀窍，我们先来开发几个例子。考虑 Carl 的家谱树节点。他的眼睛不是蓝色的，而且在家谱树中，他并没有任何（已知的）祖先，所以，对应于这个节点的家谱树并不包含蓝眼睛的人。简而言之，

```
(blue-eyed-ancestor? Carl)
```

会计算出 false。反之，由 *Gustav* 表示的家谱树中包含 Eva 节点，而 Eva 的眼睛是蓝色的。因此，

```
(blue-eyed-ancestor? Gustav)
```

会计算出 true。

这个函数的模板基本上就是 *fun-for-ftn*，只是函数的名字变成了 *blue-eyed-ancestor?*。跟往常一样，我们使用模板来指导函数的设计。首先我们假设(empty? *a-ftree*)成立。在这种情况下，家谱树是空的，于是没有人的眼睛是蓝色的。因此这时的答案必然是 false。

模板中的第二个子句包含了多个表达式，我们必须解释这些表达式：

1. （*blue-eyed-ancestor?*(child-father *a-ftree*)），该表达式在父亲的 *ftn* 中有没有蓝眼睛的人；
2. (*blue-eyed-ancestor?* (child-mother *a-ftree*))，该表达式判断在母亲的 *ftn* 中有没有蓝眼睛的人；
3. (child-name *a-ftree*)，该表达式提取出 *child* 的名字；
4. (child-date *a-ftree*)，该表达式提取出 *child* 的出生日期；
5. (child-eyes *a-ftree*)，该表达式提取出 *child* 眼睛的颜色。

现在就是使用这些值进行适当计算的时候了。显然，如果 *child* 结构体的 *eyes* 字段包含了'blue，那么函数的返回值就应是 true。否则，如果在父亲或者母亲的家谱树中有蓝眼睛的人，函数也要返回 true。其他数据是没有用的。

讨论表明应该使用一个条件表达式，其中第一个条件是：

```
(symbol=? (child-eyes a-ftree) 'blue)
```

模板中的两个递归就是另外两个条件。如果有一个条件返回 true，函数就会返回 true。else 子句返回 false。总而言之，第二个子句的答案就是表达式：

```
(cond
  [(symbol=? (child-eyes a-ftree) 'blue) true]
  [(blue-eyed-ancestor? (child-father a-ftree)) true]
  [(blue-eyed-ancestor? (child-mother a-ftree)) true]
  [else false])
```

图 14.3 中的第一个定义对其进行了概括。图中的第二个定义说明这个 cond 表达式等价于一个 or 表达式，这个 or 表达式一个一个地测试条件，直到有一个条件为 true，或者所有条件的计算结果都为 false 为止。

```
;; blue-eyed-ancestor? : ftn -> boolean
;; 判断 a-ftree 是否包含一个 child 结构体，
;; 其 eyes 字段为'blue
;; 第一个版本：使用嵌套的 cond 表达式
(define (blue-eyed-ancestor? a-ftree)
  (cond
    [(empty? a-ftree) false]
    [else (cond
            [(symbol=? (child-eyes a-ftree) 'blue) true]
            [(blue-eyed-ancestor? (child-father a-ftree)) true]
            [(blue-eyed-ancestor? (child-mother a-ftree)) true]
            [else false])]))
```

```
;; blue-eyed-ancestor? : ftn -> boolean
;; 判断 a-ftree 是否包含一个 child 结构体，
;; 其 eyes 字段为'blue
;; 第二个版本：使用 or 表达式
(define (blue-eyed-ancestor? a-ftree)
  (cond
    [(empty? a-ftree) false]
    [else (or (symbol=? (child-eyes a-ftree) 'blue)
              (or (blue-eyed-ancestor? (child-father a-ftree))
                  (blue-eyed-ancestor? (child-mother a-ftree))))]))
```

图 14.3　两个寻找蓝眼睛祖先的函数

函数 *blue-eyed-ancestor?*的不同寻常之处是，它在 cond 表达式中使用递归。要理解这是如何工作的，

我们来手工计算 *blue-eyed-ancestor?*作用于 *Carl* 的结果：

```
  (blue-eyed-ancestor? Carl)
= (blue-eyed-ancestor? (make-child empty empty 'Carl 1926 'green))
= (cond
    [(empty? (make-child empty empty 'Carl 1926 'green)) false]
    [else
      (cond
        [(symbol=?
           (child-eyes (make-child empty empty 'Carl 1926 'green))
           'blue)
         true]
        [(blue-eyed-ancestor?
           (child-father (make-child empty empty 'Carl 1926 'green)))
         true]
        [(blue-eyed-ancestor?
           (child-mother (make-child empty empty 'Carl 1926 'green)))
         true]
        [else false])])
= (cond
    [(symbol=? 'green 'blue) true]
    [(blue-eyed-ancestor? empty) true]
    [(blue-eyed-ancestor? empty) true]
    [else false])
= (cond
    [false true]
    [false true]
    [false true]
    [else false])
= false
```

计算证实了 *blue-eyed-ancestor?*对 *Carl* 能正常运行，也阐明了函数是如何工作的。

习题

习题 14.1.1 图 14.3 中 blue-eyed-ancestor?的第二个定义没有使用嵌套的条件，而是使用了 or 表达式。用手工计算证明，这个函数作用于 empty 和 *Carl* 时会返回与第一个定义相同的输出。

习题 14.1.2 用手工计算证实：

```
(blue-eyed-ancestor? empty)
```

的计算结果为 false。

分别用手工和使用 DrScheme 计算(*blue-eyed-ancestor? Gustav*)。在手工计算时，跳过那些提取、比较和涉及到 empty?条件的计算步骤。另外，尽可能重用已确定的等式，特别是上述等式。

习题 14.1.3 开发 *count-persons*，这个函数读入一个家谱树节点，返回相应家谱树中的人数。

习题 14.1.4 开发函数 *average-age*，这个函数读入一个家谱树节点和当前年份，返回家谱树中所有人的平均年龄。

习题 14.1.5 开发函数 *eye-colors*，该函数读入一个家谱树节点，返回这棵树中所有眼睛颜色的表。在该表中，一种眼睛颜色可以出现多次。

提示：使用 Scheme 的操作 append，该操作读入两个表，返回这两个表的连接。例如：

```
  (append (list 'a 'b 'c) (list 'd 'e))
= (list 'a 'b 'c 'd 'e)
```

我们会在第 17 章中讨论类似于 append 的函数的设计。

习题 14.1.6　假设我们需要函数 *proper-blue-eyed-ancestor?*（严格意义上的蓝眼睛祖先）。这个函数类似于 *blue-eyed-ancestor?*，但是它只在某个严格意义上的祖先是蓝眼睛的情况下才返回 true。也就是说，在给定节点为蓝眼睛的情况下，该函数并不一定返回 true。

这个新函数的合约与原来的函数一样：

```
;;proper-blue-eyed-ancestor? : ftn -> boolean
;;判断 a-ftree 有没有蓝眼睛的祖先
(define (proper-blue-eyed-ancestor? a-ftree) ...)
```

与原来函数的合约只是稍有不同。

为了能充分领会两个函数之间的区别，我们来观察 Eva，她是蓝眼睛的，但是她并没有蓝眼睛的祖先。因此，

```
(blue-eyed-ancestor? Eva)
```

为 **true**，但

```
(proper-blue-eyed-ancestor? Eva)
```

为 false。毕竟，*Eva* 不是她自己的（严格意义上的）祖先。

假设你的一位朋友看到了这个函数的用途说明，并给出如下程序：

```
(define (proper-blue-eyed-ancestor? a-ftree)
  (cond
    [(empty? a-ftree) false]
    [else (or (proper-blue-eyed-ancestor? (child-father a-ftree))
              (proper-blue-eyed-ancestor? (child-mother a-ftree)))]))
```

对于任何一个 *ftn A*，(*proper-blue-eyed-ancestor? A*)的返回值会是什么？

改正这位朋友的程序。

14.2　补充练习：二叉搜索树

虽然家谱树很少使用，但程序设计者经常使用树。一种非常著名的树是二叉搜索树。许多应用软件都使用二叉搜索树来存取信息。

为了使这个概念更为具体，我们来讨论管理人员信息的二叉树。在这种情况下，二叉树类似于家谱树，但是它不包含 *child* 结构体，而是包含 *node*（节点）：

```
(define-struct node (ssn name left right))
```

这里，我们决定记录人的社会保险号码、名字和另两棵树。在包含两棵树这一方面，二叉树类似于家谱树中的父亲字段和母亲字段，尽管 *node* 与其 *left* 和 *right* 树之间的关系并不是家庭关系。

相应的数据定义类似于家谱树的数据定义：

> *binary-tree*（简称 *BT*）是下列两者之一：
> 1. false；
> 2. (make-node *soc pn lft rgt*)，
>
> 其中 *soc* 是数，*pn* 是符号，*lft* 和 *rgt* 是 *BT*。

这里选用 false 来表示缺乏信息，这一点是任意的。事实上，也可以选用 empty。不过，我们更熟悉 false，而且比起其他东西，它更好用，更常见。

下面是两棵二叉树：

```
(make-node
  15
  'd
  false
  (make-node 24 'i false false))
```

```
(make-node
  15
  'd
  (make-node 87 'h false false)
  false)
```

图 14.4 是树的形象表示。树是从上向下绘制的，也就是说，树根在顶部，树冠在底部。每一个圆点代表一个节点，节点用相应的 *node* 结构体的 *ssn* 字段标出。另外，树中省略了 false。

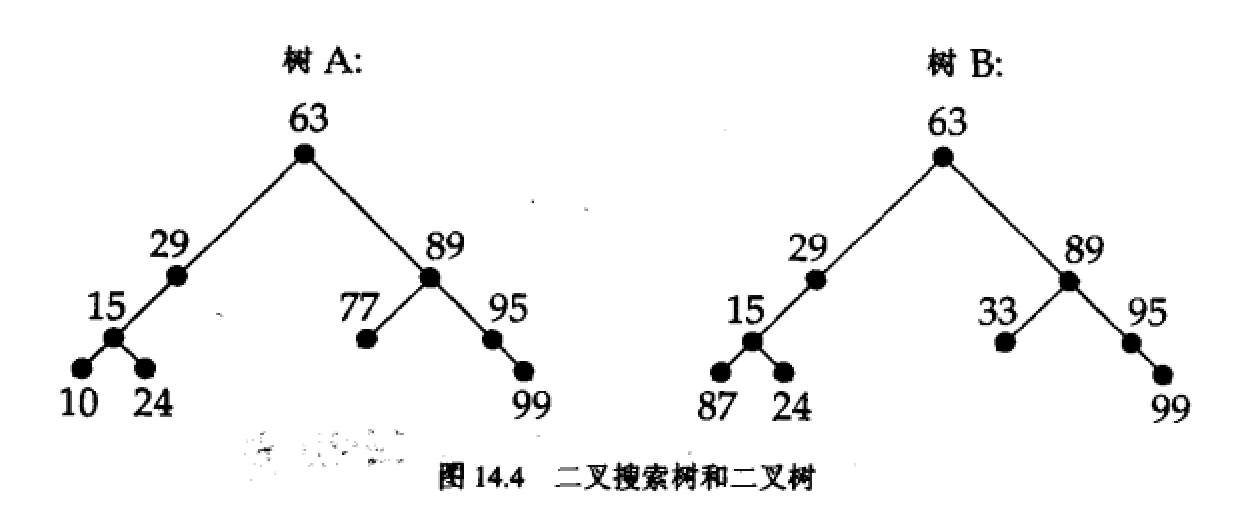


图 14.4 二叉搜索树和二叉树

习题

习题 14.2.1 按照图 14.4 中的方法，绘出上述两棵树的形象表示。然后开发 *contains-bt*，该函数读入一个数和一棵 *BT*，判断这个数是否在树中出现。

习题 14.2.2 开发 *search-bt*，该函数读入数 *n* 和一棵 *BT*。如果这棵树中包含一个 *node* 结构体，其 *soc* 字段值为 *n*，函数就返回这个节点的 *pn* 字段的值；否则，函数返回 false。

提示：使用 *contains-bt*，或者使用 boolean?求出 *search-bt* 是否成功作用于某一棵子树。本部分最后的独立章节还将讨论这第二种技术，它被称为回溯。

图 14.4 中的两棵树都是二叉树，但是它们在一个很重要的方面不同。如果从左向右读出这两棵树中的数，就可以得到两个序列：

Tree A:	10	15	24	29	63	77	89	95	99
Tree B:	87	15	24	29	63	33	89	95	99

树 A 的序列是按照升序排列的，而树 B 不是。

有序排列信息的二叉树称为二叉搜索树。所有二叉搜索树都是二叉树，但是二叉树不一定是二叉搜索树。我们说二叉搜索树是二叉树严格意义上的子类型，换句话说，二叉搜索树并不包含所有的二叉树。更具体地，我们给出一个条件——或者数据不变式——把二叉搜索树从二叉树中区分出来：

BST 不变式

binary-search-tree（二叉搜索树，简称 *BST*）是 *BT*：

1. false 总是 *BST*；
2. (make-node *soc pn lft rgt*)是 *BST*，如果：
   a. *lft* 和 *rgt* 是 *BST*，
   b. *lft* 中所有的 *ssn* 数都比 *soc* 小，
   c. *rgt* 中所有的 *ssn* 数都比 *soc* 大。

其中的第二和第三个条件与我们在以前的数据定义中所看到的不同。在构建 *BST* 时，它们提出了额外的、不寻常的要求。我们必须检查这些树中所有的数，才能确保它们都比 *soc* 小（或大）。

习题

习题 14.2.3　开发函数 *inorder*（中序），该函数读入一棵二叉树，返回树中所有 *ssn* 数组成的表。该表以从左到右的顺序（就是前面我们所用的顺序）列出这些数。

提示：使用 Scheme 的操作 append，该操作连接多个表：

```
(append (list 1 2 3) (list 4) (list 5 6 7))
```

计算为：

```
(list 1 2 3 4 5 6 7)
```

对于二叉搜索树来说，*inorder* 会返回什么？

在 *BST* 中寻找某个特定的 *node* 比在 *BT* 中寻找特定的 *node* 步骤要少。要判断某个 *BT* 是否包含含有特定 *ssn* 字段的节点，函数必须检查树中所有的 *node*。反之，检查一棵二叉搜索树所需的步骤要少得多。假设我们有 *BST*：

```
(make-node 66 'a L R)
```

如果要寻找 66，我们已经找到它了。现在，假设我们要寻找 63。在上述的 *node* 中，我们可以只搜索 *L*，因为所有 *ssn* 小于 66 的 *node* 都在 *L* 中。类似地，如果我们要寻找 99，就可以忽略 *L* 而搜索 *R*，因为所有 *ssn* 大于 66 的 *node* 都在 *R* 中。

习题

习题 14.2.4　开发 *search-bst*，该函数读入数 *n* 和一个 *BST*。如果这棵树中包含一个 *node* 结构体，其 *soc* 字段为 *n*，函数就返回这个节点的 *pn* 字段的值。否则，函数返回 false。函数必须利用 *BST* 不变式，尽可能地减少比较的次数。请比较在二叉搜索树中进行的查找和在有序表中进行的查找（习题 12.2.2）。

建立二叉树很简单，而建立二叉搜索树就是一件复杂的、易于出错的事情。要建立一棵 *BT*，我们用

make-node 把两个 *BT*、一个 *ssn* 和一个 *name* 结合起来。按照定义，这样就可以得到一个 *BT*。要建立一棵 *BST*，这样做就不行了，因为这样做所得的结果一般不是 *BST*。例如，如果某棵树包含 3 和 5，而另一棵树包含 2 和 6，我们就没有办法把这两棵 *BST* 连接成一棵二叉搜索树。

至少有两种方法可以解决这个问题。第一种方法是，给定一个由数和符号组成的表，我们可以手工求出相应的 *BST* 的外形，然后使用 make-node 建立 *BST*。第二种方法，我们可以写出一个函数，它由表建立 *BST*，该 *BST* 的形状是一个 *node* 后跟另一个 *node*。

习题

习题 14.2.5 开发函数 *create-bst*，该函数读入 *BST B*、数 *N* 和符号 *S*，返回一棵 *BST*，该 *BST* 类似于 *B*，但是在（原来）某个 false 子树的位置上包含下面的 *node* 结构体：

```
(make-node N S false false)
```

用(*create-bst* false 66 'a)来测试这个函数；这样做应该会建立一个单独的 *node*。接下来证明如下等式成立：

```
  (create-bst (create-bst false 66 'a) 53 'b)
= (make-node 66
             'a
             (make-node 53 'b false false)
             false)
```

最后，用 *create-bst* 建立图 14.4 中的树 A。

习题 14.2.6 开发函数 *create-bst-from-list*，这个函数读入一个由数和名字组成的表，反复调用 *create-bst*，返回一个 *BST*。

数和名字组成的表的数据定义如下：

> *list-of-number/name*（数和名字组成的表）是下列两者之一：
> 1. **empty**
> 2. **(cons (list *ssn nom*) *lonn*)**
>
> 其中 *ssn* 是数，*nom* 是符号，*lonn* 是 *list-of-number/name*。

考虑下面的例子：

```
(define sample          (define sample
  '((99 o)                (list (list 99 'o)
    (77 l)                      (list 77 'l)
    (24 i)                      (list 24 'i)
    (10 h)                      (list 10 'h)
    (95 g)                      (list 95 'g)
    (15 d)                      (list 15 'd)
    (89 c)                      (list 89 'c)
    (29 b)                      (list 29 'b)
    (63 a)))                    (list 63 'a)))
```

它们是相等的，尽管左边是用引号缩写定义的，而右边是用 list 定义的。把 *create-bst-from-list* 作用于这个表，就会得到图 14.4 中左侧的树。

14.3　表中的表

环球网（World Wide Web）已成为国际互联网——一个全球性的计算机网络——中最有意义的部分了。简单说来，环球网是网页的集合。每个网页都是文字、图片、电影、音频信息以及许多其他东西的序列。更重要的是，网页还可以包含到其他网页的链接。

人们使用网络浏览器来浏览网页。浏览器把网页表示为文字、图像等的序列。网页中的某些文字下方会有下划线，单击带下划线的文字可以打开一个新的网页。许多高级浏览器还提供网页编辑器，以帮助人们创建网页。简而言之，我们应当能够在计算机上表示网页，而且应该有许多处理网页的函数。

为了简化问题，我们只考虑包含文字及嵌套网页的网页。理解这类网页的一种方法是把它看成文字和网页的序列。这种非正式的描述表明，网页的一种自然表示法是表，表中包含符号和网页，其中符号代表单词，网页代表嵌套网页。毕竟，我们以前曾强调过，表中可以包含不同类型的东西。不过，当使用数据定义来表示这种思想的时候，我们得到了一种相当不同寻常的东西：

> *Web-page*（网页，简称 *WP*）是下列三者之一：
>
> 1. **empty**；
> 2. **(cons** *s wp***)** ，
>
>   其中 *s* 是符号，*wp* 是网页；
> 3. **cons** *ewp wp*),
>
>   其中 *ewp* 和 *wp* 都是网页。

该数据定义与符号表定义的不同之处是，它有三个子句，而不是两个，而且它有三个自引用，而不是一个。在这些自引用中，最不寻常的就是在 *cons* 构造的表开始的那一个。我们把这样的网页称为直接嵌入的网页。

因为这个数据定义比较不寻常，所以在继续之前，我们先来构造一些网页的例子。下面是一个普通网页：

```
'(The TeachScheme! Project aims to improve the
  problem-solving and organization skills of high
  school students. It provides software and lecture
  notes as well as exercises and solutions for teachers.)
```

它只包含单词。下面是一个复杂网页：

```
'(The TeachScheme Web Page
  Here you can find:
  (LectureNotes for Teachers)
  (Guidance for (DrScheme: a Scheme programming environment))
  (Exercise Sets)
  (Solutions for Exercises)
  For further information: write to scheme@cs)
```

直接嵌入的网页由括号和符号'LectureNotes、'Guidance、'Exercises 及'Solutions 开始。第二个嵌入的网页还包含了另一个嵌入的网页，由单词'DrScheme 开始。称该网页是嵌入于整个网页的网页。

现在来开发函数 *size*，该函数读入一个网页，返回其自身以及所有嵌入网页所包含的单词数：

```
;; size : WP -> number
;; 计算在 a-wp 中出现的符号数
```

```
(define (size a-wp) ...)
```

前面的两个网页就是两个很好的例子，不过它们过于复杂了。下面是三个例子，每个例子对应于一种数据子类型：

```
(= (size empty)
   0)
(= (size (cons 'One empty))
   1)
(= (size (cons (cons 'One empty) empty))
   1)
```

前两个例子是显然的。第三个例子需要稍作解释，函数的参数是一个网页，其中只包含一个直接嵌入网页，这个直接嵌入网页就是第二个例子中的网页，所以第三个例子只包含一个符号。

要开发 *size* 的模板，可按照设计诀窍仔细执行每个步骤。数据定义的形状说明我们需要三个 cond 子句：一个子句处理 empty 页，一个子句处理由符号开始的页，另一个子句处理由嵌入的网页开始的页。虽然第一个测试 empty 的条件我们已经很熟悉了，但是第二和第三个条件需要进一步地检查，因为在数据定义中，这两个子句都使用了 cons，所以简单地使用 cons?并不能区分这两种数据形式。

如果网页不是 empty，那么它必然是 cons 结构，后两种数据形式之间的特征是表中的第一个元素。换句话说，第二个条件必须使用一个测试 *a-wp* 的第一个元素的谓词：

```
;; size : WP -> number
;; 计算在 a-wp 中出现的符号数
(define (size a-wp)
  (cond
    [(empty? a-wp) ...]
    [(symbol? (first a-wp)) ... (first a-wp) ... (size (rest a-wp)) ...]
    [else ... (size (first a-wp)) ... (size (rest a-wp)) ...]))
```

模板的其余部分就很一般了。第二和第三个 cond 子句中包含了表的第一个元素和其余部分的选择器表达式。因为(rest *a-wp*)总是网页，而在第三个子句中，(first *a-wp*)也是网页，所以我们还为这些选择器表达式加上 *size* 的递归调用。

使用例子和模板，我们就可以开发出 *size*：参见图 14.5。模板和定义之间没有很大的区别，这再次表明，我们只需系统地思考函数的输入数据定义，就可以设计出函数相当大的部分。

```
;; size : WP -> number
;; 计算在 a-wp 中出现的符号数
(define (size a-wp)
  (cond
    [(empty? a-wp) 0]
    [(symbol? (first a-wp)) (+ 1 (size (rest a-wp)))]
    [else (+ (size (first a-wp)) (size (rest a-wp)))]))
```

图 14.5 网页的 size 的定义

习题

习题 14.3.1 简要地说明如何使用模板和例子来定义 *size*。使用前述的例子对 *size* 进行测试。

习题 14.3.2 开发函数 *occurs1*，该函数读入一个网页和一个符号，返回该符号在网页中出现的次数，忽略嵌入的网页。

开发函数 *occurs2*，该函数类似于 *occurs1*，但是它计算该符号所有的出现次数，包括在嵌入网页中的出现。

习题 14.3.3　开发函数 *replace*，该函数读入符号 *new* 和 *old*，以及网页 *a-wp*，返回一个网页，该网页在结构上和 *a-wp* 相同，但其中所有 *old* 的出现都被替换成 *new*。

习题 14.3.4　人们并不喜欢深层的网页，因为要在这种网页中寻找有用的信息需要进行太多的网页切换。因为这个原因，网页设计者还需要测量某个网页的深度。只包含符号的网页的深度为 0；包含一个直接嵌入页的网页深度为该嵌入页的深度加 1。如果某个网页包含多个直接嵌入页，它的深度就是最深的嵌入页的深度加 1。开发 *depth*，该函数读入一个网页，并计算它的深度。

14.4　补充练习：Scheme 求值

DrScheme 自身是一个程序，它由多个部分组成。其中一个功能是检查定义及表达式是否合乎语法，另一个功能是计算 Scheme 表达式。使用在这一章中所学的知识，我们现在可以开发这些函数的简化版本。

我们的第一个任务是约定 Scheme 程序的数据表示法。换一种说法，我们必须解决如何用 Scheme 数据来表示 Scheme 表达式的问题。这听起来很奇怪，但做起来不难。假设我们一开始只需要表示数、变量、加法和乘法。显然，数可以表示数，符号可以表示变量。不过，表示加法和乘法需要使用复合数据类型，因为它们由一个算子和两个子表达式组成。

一种直接表示加法和乘法的方法是使用两个结构体：一个结构体表示加法，另一个表示乘法。下面就是结构体的定义：

```
(define-struct add (left right))
(define-struct mul (left right))
```

每个结构体都包含两个成分，一个成分代表操作左边的表达式，另一个代表右边的表达式。

我们来看一些例子：

Scheme 表达式	Scheme 表达式的表示法
3	3
x	*x*
(* 3 10)	(make-mul 3 10)
(+ (* 3 3) (* 4 4))	(make-add (make-mul 3 3) (make-mul 4 4))
(+ (* *x x*) (* *y y*))	(make-add (make-mul 'x 'x) (make-mul 'y 'y))
(* 1/2 (* 3 3))	(make-mul 1/2 (make-mul 3 3))

这些表达式覆盖了所有的情形：数、变量、简单表达式以及嵌套的表达式。

习题

习题 14.4.1　给出 Scheme 表达式的表示法的数据定义，然后把下列表达式转化成该表示法：

```
1. (+ 10 -10)
2. (+ (* 20 3) 33)
3. (* 3.14 (* r r))
```

```
4. (+ (* 9/5 c) 32)
5. (+ (* 3.14 (* o o)) (* 3.14 (* i i)))
```

Scheme 求值程序是一个函数，该程序读入一个 Scheme 表达式的表示法，并返回它的值。例如，表达式 3 的值是 3，(+ 3 5)的值是 8，(+ (* 3 3) (* 4 4)) 的值是 25，等等。因为我们现在没有考虑定义，所以包含变量的表达式是没有值的，例如(+ 3 *x*)就没有值；毕竟，我们不知道这个变量代表了什么。换一种说法，我们的 Scheme 求值程序只作用于不包含变量的表达式。我们称这种表达式是数值的。

习题 14.4.2 开发函数 *numeric?*，该函数读入一个 Scheme 表达式（的表示法），判断它是不是数值的。

习题 14.4.3 给出数表达式的数据定义。开发函数 *evaluate-expression*，该函数读入一个 Scheme 表达式（的表示法），计算它的值。在完成这个函数的测试之后，修改它，使它可以读入所有类型的 Scheme 表达式；如果修改后的函数遇到一个变量，它就产生一个错误信息。

习题 14.4.4 人们在计算调用(*fa*)时，会用 *a* 代替 *f* 的参数。更一般地说，人们在计算带变量的表达式时，会把变量替换成值。

开发函数 *subst*，该函数读入变量（的表示法）*V*、数 *N* 以及一个 Scheme 表达式（的表示法），它返回一个结构相等的表达式，把其中所有 *V* 的出现都替换为 *N*。



相互引用的数据定义



在前一章中，我们开发了家谱树、网页和 Scheme 表达式的数据表示法。按照完全相同的设计诀窍，我们可以开发处理这些数据定义的函数。如果要更为现实地表示网页或 Scheme 表达式，或者要研究后代家谱树而不是祖先树，我们就必须学习描述相互关联的数据类型。也就是说，如果数据定义不仅引用它们自身，还引用其他的数据定义，我们必须同时给出多个数据定义。

15.1 由结构体组成的表与结构体中的表

在用追溯形式建立家谱树时，我们通常从某个后代出发，依次处理他的父母、祖父母，等等。而构建树时，我们会不断添加谁是谁的孩子，而不是写出谁是谁的父母，从而建立一棵后代家谱树。

绘制后代树的过程和绘制祖先树一样，只是所有箭头的方向都反了过来。图 15.1 用后代的观点给出了与图 14.1 一样的家谱树。

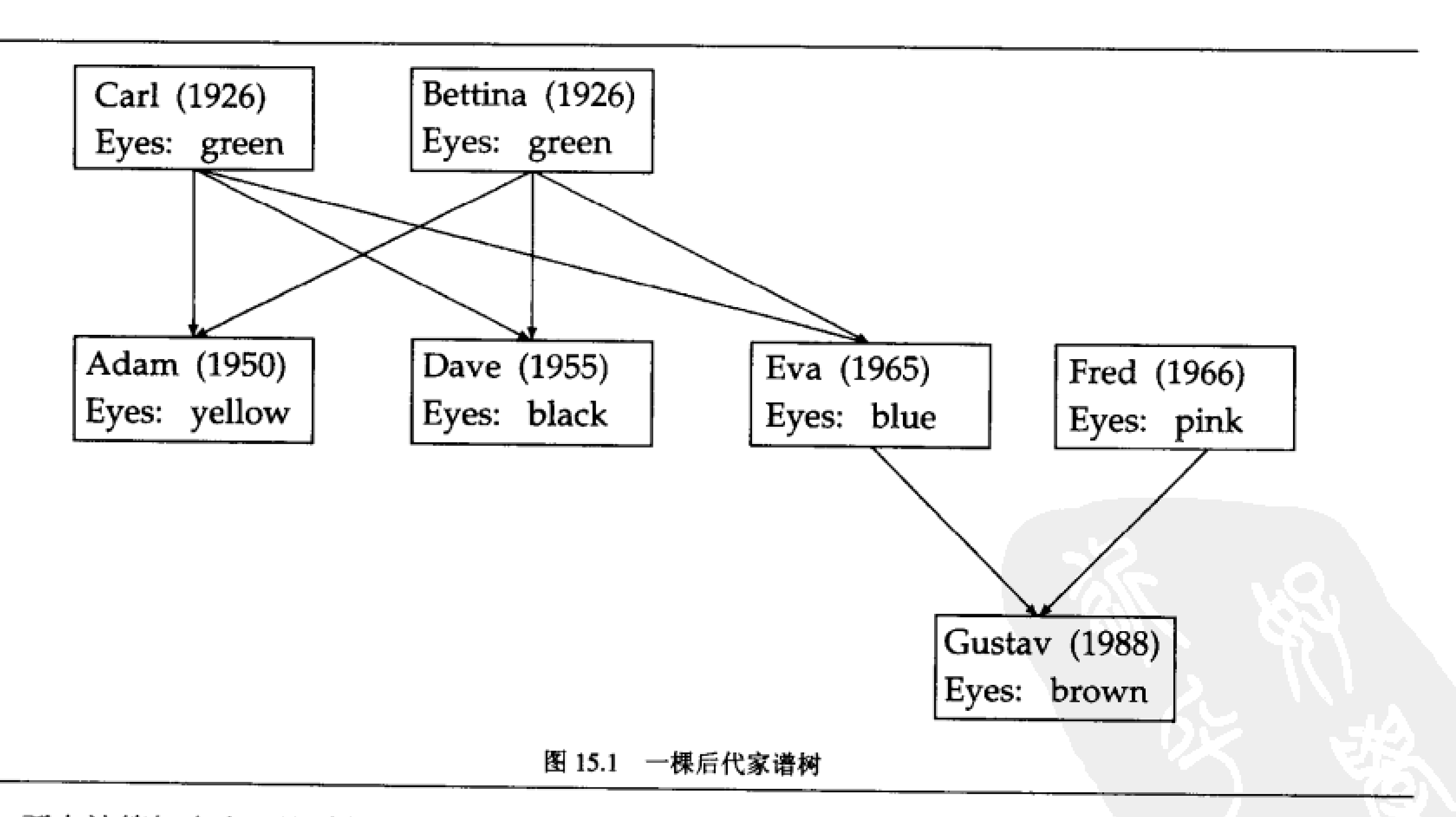


图 15.1 一棵后代家谱树

要在计算机中表示这种新的家谱树以及它们的节点，需要使用与祖先树不同的数据类型。这一次，节点中必须包含孩子的信息，而不是两位父母的信息。下面是结构体的定义：

```
(define-struct parent (children name date eyes))
```

parent（父母）结构体中的后三个字段包含个人的某些基本信息，与对应的 *child* 结构体一样，但是这第一个字段的内容有一个有趣的问题。既然一对父母可以有任意数量的孩子，*children* 字段必须要包

含数量不定的节点，每个节点代表一个孩子。

自然的选择是令 *children* 代表由 *parent* 结构体组成的表，这个表代表孩子；如果某人没有孩子，该表就是 empty。这表明了如下的数据定义：

> ***parent* 是结构体：**
>
> (make-parent *loc n d e*)
>
> 其中 *loc* 是孩子的表，*n* 和 *e* 是符号，*d* 是数。

不幸的是，这个数据定义违反了我们关于定义的标准。具体说来，它提到了一个还没有定义过的集合：孩子的表。

既然无法在不知道孩子的表是什么的情况下定义父母类型，我们就先定义孩子的表：

> *list of children*（孩子的表）是下列两者之一：
>
> 1. **empty**
> 2. (**cons** *p loc*)，其中 *p* 是 parent，*loc* 是孩子的表。

这第二个定义看上去是标准的，但是它遇到了与 *parent* 一样的问题，它所引用的未知类型是父母类型，而父母类型在没有孩子的表的定义时也不能定义，以此类推。

结论是，这两个定义相互引用对方，它们只在*同时*定义的情况下才有意义：

> *parent* 是结构体：
>
> (**make-parent** *loc n d e*)
>
> 其中 *loc* 是孩子的表，*n* 和 *e* 是符号，*d* 是数。
>
> *list-of-children*（孩子的表）是下列两者之一：
>
> 1. **empty**
> 2. (**cons** *p loc*)，其中 *p* 是 **parent**，而 *loc* 是孩子的表。

如果两个（或更多）数据定义相互应用，我们就称它们为相互引用的。

现在，我们可以把图 15.1 中的家谱树转化成 Scheme 表达式了。当然，在建立 *parent* 结构体之前，我们必须先定义所有表示其孩子的节点。同第 14.1 节一样，最好的方法是，在使用某个 *parent* 结构体之前先给它命名，下面是一个例子：

```
(define Gustav (make-parent empty 'Gustav 1988 'brown))
(make-parent (list Gustav) 'Fred 1950 'yellow)
```

要建立 Fred 的 *parent* 结构体，我们先定义 Gustav 的结构体，这样就可以用(list *Gustav*)表示 Fred 的孩子。

图 15.2 给出了这颗后代树的完整 Scheme 表示。为了避免重复，其中还包括了孩子的表的定义。请比较这个定义和图 14.2 中同一个家族的祖先树表示。

```
;; 年轻一代人:
(define Gustav (make-parent empty 'Gustav 1988 'brown))

(define Fred&Eva (list Gustav))

;; 中间一代人:
(define Adam (make-parent empty 'Adam 1950 'yellow))
(define Dave (make-parent empty 'Dave 1955 'black))
(define Eva (make-parent Fred&Eva 'Eva 1965 'blue))
(define Fred (make-parent Fred&Eva 'Fred 1966 'pink))

(define Carl&Bettina (list Adam Dave Eva))

;; 老一代人:
(define Carl (make-parent Carl&Bettina 'Carl 1926 'green))
(define Bettina (make-parent Carl&Bettina 'Bettina 1926 'green))
```

图 15.2 后代家谱树的 Scheme 表示

现在我们来研究 *blue-eyed-descendant?*的开发，它是 *blue-eyed-ancestor?*的自然对应物。该函数读入一个 *parent* 结构体，判断他或者他的任何一个后代的眼睛是不是蓝色的：

```
;; blue-eyed-descendant? : parent -> boolean
;; 判断 a-parent 或者他的任何意个后代（孩子、孙子等）
;; 的 eyes 字段中是否包含'blue
(define (blue-eyed-descendant? a-parent) ...)
```

下面是三个简单的例子，以测试的形式给出：

```
(boolean=? (blue-eyed-descendant? Gustav) false)
(boolean=? (blue-eyed-descendant? Eva) true)
(boolean=? (blue-eyed-descendant? Bettina) true)
```

只需观察图 15.1 就可以解释每一个例子的答案。

按照规则，*blue-eyed-descendant?*的模板相当简单。因为这个函数的输入是普通的结构体，模板就只包含选择器表达式，用来提取结构体中的字段：

```
(define (blue-eyed-descendant? a-parent)
  ... (parent-children a-parent) ...
  ... (parent-name a-parent) ...
  ... (parent-date a-parent) ...
  ... (parent-eyes a-parent) ... )
```

parent 的结构体定义指定了四个字段，所以模板中有四个表达式。

模板中的表达式提醒我们，*parent* 的眼睛颜色是可用的，而且也应该被检查，因此我们加上一个 cond 表达式，比较(**parent-eyes** *a-parent*)和'blue：

```
(define (blue-eyed-descendant? a-parent)
  (cond
    [(symbol=? (parent-eyes a-parent) 'blue) true]
    [else
      ... (parent-children a-parent) ...
      ... (parent-name a-parent) ...
```

```
        ... (parent-date a-parent) ...]))
```

如果这个条件成立，答案就是 true。else 子句中包含了其他的表达式。对于眼睛颜色来说，*name* 和 *date* 字段是没有用的，所以我们可以忽略这两个表达式。这样，所剩下的就是：

```
(parent-children a-parent)
```

该表达式从 *parent* 结构体中提取出孩子的表。

如果某个 *parent* 结构体的眼睛颜色不是'blue，我们就必须在孩子的表中搜索蓝眼睛后代。按照复杂函数设计原则，我们需要向函数清单中添加一个函数，然后继续开发函数。这个要放入函数清单的函数读入一个孩子的表，检查他们或者他们的子孙中有没有蓝眼睛的人。下面是该函数的合约、头部以及用途说明：

```
;; blue-eyed-children? : list-of-children -> boolean
;; 判断 aloc 中任何一个结构体是不是蓝眼睛的,
;; 或者有没有蓝眼睛的后代
(define (blue-eyed-children? aloc) ...)
```

使用 *blue-eyed-children?*，我们就可以完成 *blue-eyed-descendant?*的定义：

```
(define (blue-eyed-descendant? a-parent)
  (cond
    [(symbol=? (parent-eyes a-parent) 'blue) true]
    [else (blue-eyed-children? (parent-children a-parent))]))
```

换句话说，如果 *a-parent* 的眼睛不是蓝色的，还需检查他的孩子的表。

在对 *blue-eyed-descendant?* 进行测试之前，我们必须先定义函数清单中的函数。为了构造 *blue-eyed-children?*的例子和测试，我们使用图 15.2 中 *list-of-children* 的定义：

```
(not (blue-eyed-children? (list Gustav)))
(blue-eyed-children? (list Adam Dave Eva))
```

Gustav 的眼睛不是蓝色的，他也没有后代的记录。因此对于(list *Gustav*)，*blue-eyed-children?*返回 false。与此相反，*Eva* 的眼睛是蓝色的，因此对于第二个孩子的表，*blue-eyed-children?*返回 true。

既然 *blue-eyed-children?*的输入是表，其模板就应是标准的模式：

```
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) ...]
    [else
      ... (first aloc) ...
      ... (blue-eyed-children? (rest aloc)) ...]))
```

接下来需要考虑两种情况：如果 *blue-eyed-children?*的输入是 empty，其答案就是 false；否则，得到下面两个表达式：

1. (first *aloc*)，该表达式从表中提取出第一个元素，即一个 *parent* 结构体；

2. (*blue-eyed-children?* (rest *aloc*))，该表达式判断 *aloc* 其余部分的结构体中有没有蓝眼睛的人或者蓝眼睛的后代。

幸运的是，我们已经有了一个函数，*blue-eyed-descendant?*它能判断某个 *parent* 结构体是不是蓝眼睛的，或者其后代中有没有蓝眼睛。这表明我们要检查：

```
(blue-eyed-descendant? (first aloc))
```

成立与否。如果成立，*blue-eyed-children?*就可以返回 true；如果不成立，第二个表达式对其余部分进行检查。

图 15.3 给出了 *blue-eyed-descendant?*和 *blue-eyed-children?*的完整定义。与其他函数群体不同，这两个函数相互引用，是相互递归的。毫不令人惊奇的是，函数定义中的相互引用对应于数据定义中的相互引用。这张图中还给出了另一对函数定义，它们使用 or 而不是嵌套的 cond 表达式。

```
;; blue-eyed-descendant? : parent -> boolean
;; 判断 a-parent 或者他的任何一个后代（孩子、
;; 孙子等）的 eyes 字段中是否包含'blue
(define (blue-eyed-descendant? a-parent)
  (cond
    [(symbol=? (parent-eyes a-parent) 'blue) true]
    [else (blue-eyed-children? (parent-children a-parent))]))

;; blue-eyed-children? : list-of-children -> boolean
;; 判断 aloc 中任何一个结构体是不是蓝眼睛的，
;; 或者有没有蓝眼睛的后代
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) false]
    [else
      (cond
        [(blue-eyed-descendant? (first aloc)) true]
        [else (blue-eyed-children? (rest aloc))])]))
```

```
;; blue-eyed-descendant? : parent -> boolean
;; 判断 a-parent 或者他的任何一个后代（孩子、
;; 孙子等）的 eyes 字段中是否包含'blue
(define (blue-eyed-descendant? a-parent)
  (or (symbol=? (parent-eyes a-parent) 'blue)
      (blue-eyed-children? (parent-children a-parent))))

;; blue-eyed-children? : list-of-children -> boolean
;; 判断 aloc 中任何一个结构体是不是蓝眼睛的，
;; 或者有没有蓝眼睛的后代
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) false]
    [else (or (blue-eyed-descendant? (first aloc))
              (blue-eyed-children? (rest aloc)))]))
```

图 15.3　两个寻找蓝眼睛后代的程序

习题

习题 15.1.1　手工计算(*blue-eyed-descendant? Eva*)，然后计算(*blue-eyed-descendant? Bettina*)。

习题 15.1.2　开发函数 *how-far-removed*，如果存在蓝眼睛后代的话，该函数判断给定的 *parent* 离蓝眼睛的后代有多远。如果给定的 *parent* 就是蓝眼睛的，这个距离就是 0；如果他不是蓝眼睛的，但是他的某个孩子是蓝眼睛的，这个距离就是 1；以此类推。如果给定的 *parent* 没有蓝眼睛后代，函数就返回 *false*。

习题 15.1.3　开发函数 *count-descendants*，该函数读入一个 *parent*，返回其后代（包括该 *parent*）的数量。

开发函数 *count-proper-descendants*，该函数读入一个 *parent*，返回其严格意义上的后代数量，也就是家谱树中不包括该 *parent* 的节点数。

习题 15.1.4　开发函数 *eye-colors*，该函数读入一个 *parent*，返回树中所有眼睛颜色的表。在该表

中，一种眼睛的颜色可以出现多次。

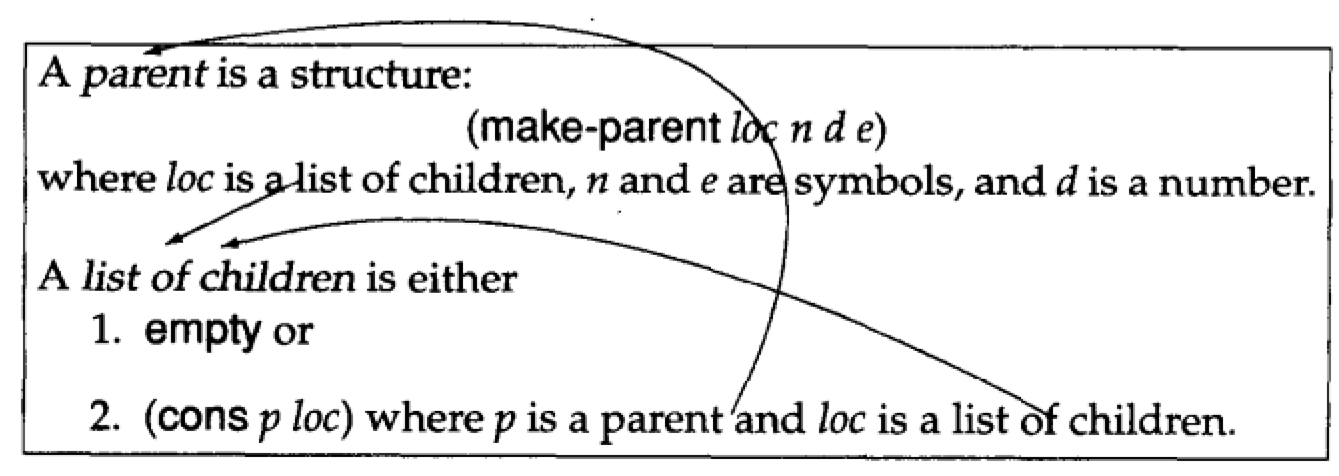
提示：使用 Scheme 的操作 append，该操作读入两个表，返回这两个表的连接。

15.2 为相互引用的定义设计函数

对自引用数据定义的函数设计诀窍一般化，就可得到相互引用数据定义的函数设计诀窍。事实上，为了处理相互引用的数据定义，只需要增加两条建议。第一条，我们必须同时建立多个模板，每个模板对应一个数据定义；第二条，我们必须用自引用和相互引用来注释模板，所谓相互引用，就是不同模板之间的相互引用。下面是对设计诀窍的差别更详细的解释：

数据分析和设计：如果问题中提到了许多不同的（任意大）信息类型，就需要一组自引用的或者相互引用的数据定义。在这组数据定义中，找出自引用和相互引用。

在上述的例子中，我们需要两个相关的定义：



A *parent* is a structure:

(make-parent *loc n d e*)

where *loc* is a list of children, *n* and *e* are symbols, and *d* is a number.

A *list of children* is either

1. empty or
2. (cons *p loc*) where *p* is a parent and *loc* is a list of children.

前一个定义是关于父母的，后一个定义是关于孩子的表的。前一个定义（无条件地）用符号、数和孩子的表定义了父母，也就是说，它包含了对第二个定义的相互引用。第二个定义是一个条件定义。它的第一个子句很简单；它的第二个子句引用了 *parent* 以及 *list-of-children* 的定义。

合约、用途说明、头部：要处理相互联系的数据类型，需要和数据定义数量一样多的函数。因此，我们必须并行地给出和数据定义一样多的合约、用途说明以及头部。

模板：模板须遵循关于复合数据、混合数据以及自引用数据的建议并行地建立。最后，我们必须确定，对于每一个选择器表达式，它是不是对应某个相互引用的定义。如果是这样，我们用某种说明相互引用的方法对它进行注释。

下面是例子的模板：

```
(define (fun-parent a-parent)
  ... (parent-name a-parent) ...
  ... (parent-date a-parent) ...
  ... (parent-eyes a-parent) ...
  ... (fun-children (parent-children a-parent)) ...)

(define (fun-children aloc)
  (cond
    [(empty? aloc) ...]
    [else ... (fun-parent (first aloc)) ... (fun-children (rest aloc)) ...]))
```

fun-parent 模板中没有条件，因为 *parent* 的数据定义中并不包含任何子句，而是包含对第二个模板的相互引用：处理 *parent* 结构体的 *children* 字段。按照同样的规则，*fun-children* 是一个条件式，第二个 cond 子句中包含了一处自引用，处理表的 rest 部分，以及对表的 first 元素——即 *parent* 结构体——的一处相互引用。

对数据定义和模板的比较表明了它们是多么的类似。为了强调自引用和相互引用的相似性，我们用箭头对数据定义和模板进行注释。不难看出，在两张图中相应的箭头有着相同的起源和相同的目标。

主体：在开始创作最后的定义时，我们要从一个模板或 cond 子句开始，这个模板中应该不包含自引用以及对其他模板的相互引用。对于这样的模板或 cond 子句，结果一般较为易于给出。

函数主体设计的其余步骤跟以前一样。在处理其他的子句或函数的时候，要提醒自己模板中的表达式计算出什么，假设所有函数都已经能按照合约的描述运作。接下来应该决定怎样把这些数据结合成最终的答案。在这样做的时候，还必须记住关于复杂函数设计的原则（参见第 7.3 节以及第 12 章）。

图 15.4 总结了扩展的设计诀窍。

阶段	目标	任务
数据分析和设计	给出一组相关的数据定义	开发一组相互递归的数据定义 • 至少一个定义或定义中的一个选项必须引用基本数据 • 显式地识别数据定义间的所有引用
模板	给出一组函数框架	同时开发与数据定义一样多的模板： • 遵照复合数据和/或混合数据的规则正确开发每一个模板。 • 根据数据定义中的（相互）引用，用递归和相互调用注释模板
主体	定义一组函数	给出每一个模板和模板中 **cond** 子句的 Scheme 表达式： • 解释模板中的每个表达式计算出什么。 • 在需要时，使用额外的辅助函数

图 15.4 为一组数据定义设计一组函数

基本步骤：其他的步骤请参见图 2.2、图 6.5 及图 7.3

15.3 补充练习：网页再谈

有了相互引用的数据定义，就可以用比第 14.3 节更准确的方式描述网页。下面是基本的结构体定义：

```
(define-struct wp (header body))
```

这两个字段分别是网页中的两种基本数据项：header（头部）和 body（主体）。数据定义说明主体是单词和网页的表：

> *Web-page*（网页，简称 *WP*）是结构体：
> **(make-wp** *h p***)**
> 其中 *h* 是符号，而 *p* 是（*web*）文档。

> (Web) *document*（文档）是下列三者之一：
> 1. **empty**，
> 2. **(cons** *s p***)**，
>   其中 *s* 是符号，*p* 是文档，
> 3. **(cons** *w p***)**，
>   其中 *w* 是网页，而 *p* 是文档。

习题

习题 15.3.1 开发函数 *size*，该函数读入一个网页，返回其中包含的符号（单词）数。

习题 15.3.2 开发函数 *wp-to-file*，该函数读入一个网页，返回符号表。该表包含网页主体中所有单词以及嵌入网页的所有头部，而忽略直接嵌入网页的主体。

习题 15.3.3 开发函数 *occurs*，该函数读入一个符号和一个网页，判断前者有没有在后者中出现，包括在后者的嵌入网页中出现。

习题 15.3.4 开发函数 *find*，这个函数读入一个网页和一个符号。如果该符号没有在该网页和它的嵌入网页中出现，函数就返回 false。如果该符号至少出现了一次，函数就返回通往该符号的路上所遇到的头部组成的表。

提示：定义一个类似于 *find* 的辅助函数，它仅在网页包含所需单词时返回 true。使用这个函数定义 *find*。或者，使用 boolean?来判断 *find* 的某个递归调用是返回了表还是布尔值，然后再计算结果。我们会在本部分最后的独立章节中讨论这第二种技术，它被称为回溯。





反复精化设计

在设计真正的函数时，常常会遇到这样的任务，要求设计复杂形式信息的数据表示法。完成这种任务最好的方法是使用一种著名的科学方法：反复精化。科学家们使用数学来表示真实世界，他们努力所得的结果称为模型。科学家们会使用多种方法测试模型，特别是使用模型来预测世界的属性。如果模型真的描述了真实世界的本质，那么这样作出的预言就是准确的；否则，在预言和实际结果之间就会有矛盾。例如，某位物理学家可能用一个点来表示喷气式飞机，然后使用牛顿方程预测它的运动轨迹为一条直线。后来，如果需要求飞机所受的摩擦力，该物理学家可能会在模型中加上飞机的轮廓线，用来表示其外形。一般来说，科学家会改进模型，重新测试它的有效性，直至模型充分准确为止。

程序设计者或者计算机科学家应该进行和科学家一样的行动。既然数据表示法在程序设计者的工作中起了主导作用，问题的关键就是找出真实世界信息的精确数据表示法。在复杂情形下，要做到这一点的最好方法就是反复设计表示法，从问题的基本元素开始，在充分理解当前模型后，再添加问题的更多特征。

本书已经在许多补充练习中使用了反复精化。例如，移动图形的练习从简单的圆和矩形开始；后来，开发了移动整个图形的程序。类似地，我们先以单词和嵌入网页表的形式引入了网页；在第 15.3 节中，我们改进了嵌入网页的表示法。不管怎样说，对于所有这些练习，改进都是建立在表示法上的。

这一章举例说明反复精化是程序开发的原则。本章的目标是建立文件系统的模型。文件系统是计算机的一个组成部分，它负责在计算机关闭的时候保存程序和数据。我们先详细讨论文件，再反复开发三种数据表示法。本章的最后部分是最终模型的一些编程习题。在以后的章节中，我们还会用到反复精化。

16.1 数据分析

在关闭计算机的时候，应该把处理过的函数和数据保存起来。不然的话，再一次打开计算机时就不得不再次输入所有的东西。计算机把需要长时间保存的东西放在文件中。文件是若干数据的序列。就我们的用途而言，文件就像是表。我们忽略为什么计算机要永久地存储文件，以及它是怎样永久存储文件的。

对我们来说更重要的是，在大多数计算机系统上，文件的集合是以目录[1]的形式组织的。简单地说，目录中包含了一些文件以及其他一些目录。包含在目录中的目录被称为子目录，子目录又可以包含更多的子目录和文件，以此类推。整个文件的集合被称为文件系统，或者目录树。

图 16.1 给出了一棵小型目录树的大略图形[2]。这棵树的根目录是 TS。根目录包含了一个文件（名为 read!）和两个子目录，(名为 Text 和 Libs)。前一个子目录，即 Text，包含三个文件；后一个子目录，即 Libs，包含两个子目录，每个子目录中又包含文件。图中的每个方框都有注解，目录的注解是 DIR，而文件的注解是一个数，表示文件的长度。TS 总共包含了七个文件和五个（子）目录。

[1] 在某些计算机中，目录被称为文件夹。
[2] 这张图解释了为什么计算机科学家把目录称为目录树

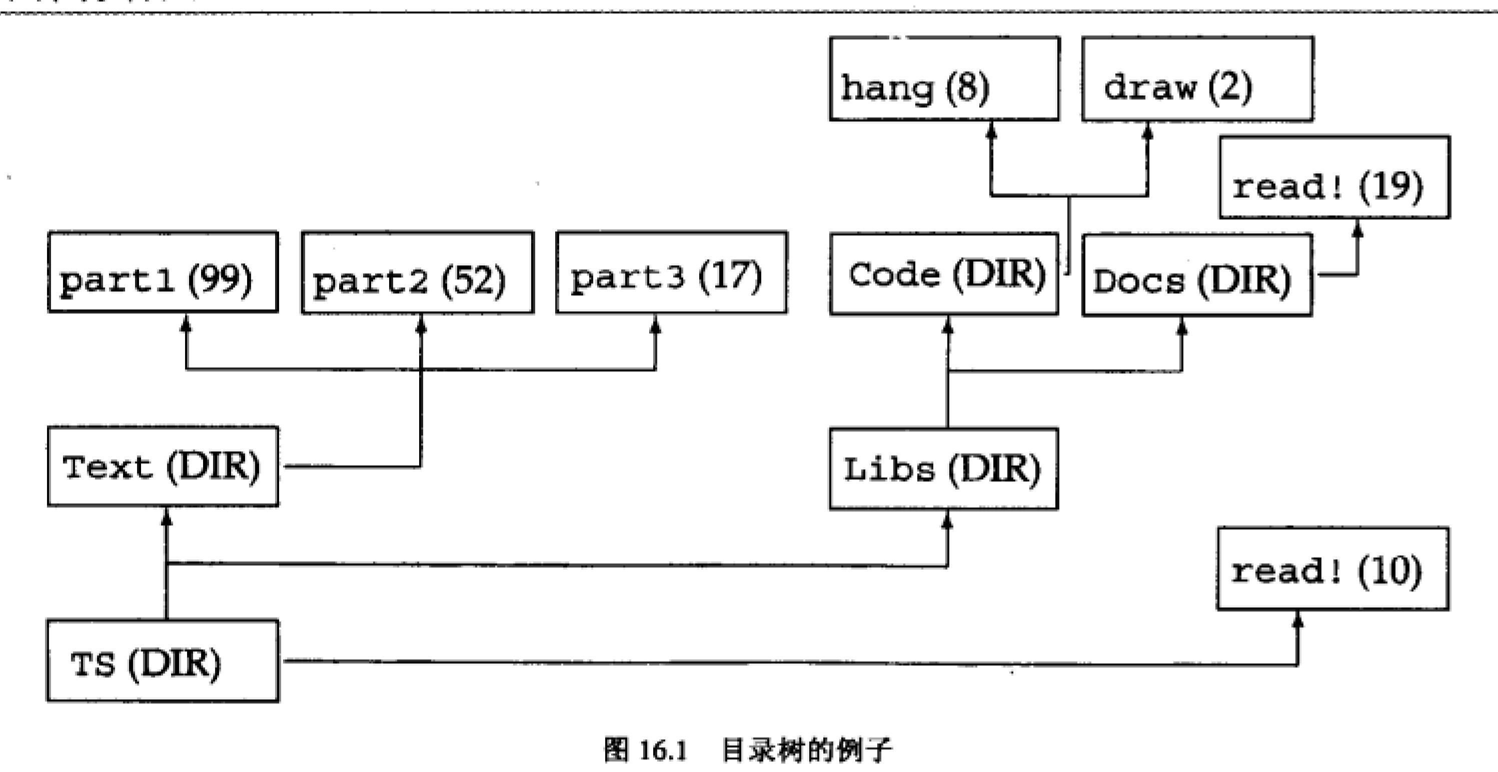


图 16.1 目录树的例子

习题

习题 16.1.1 在目录树 TS 中，文件名 read!共出现了多少次？树中所有文件的总长度是多少？树的深度是多少（它包含了多少层）？

16.2 定义数据类型，再改进它们

我们使用反复精化的方法来开发文件系统的数据表示法。需要做的第一个决定是，该把注意力集中在哪里，又该忽略什么东西。

考虑图 16.1 中的目录树，我们来想象它是怎样建立起来的。用户第一次建立目录的时候，它是空的。随着时间的推移，用户不断地添加文件和目录。一般来说，用户会用文件名来引用文件，而把目录看作容器。

模型一：思考表明，我们的第一个模型，也就是最原始的模型应该把文件当作基本实体，比方说一个代表文件名的符号，而把目录当作容器。更具体地说，我们应该把目录看作包含文件和目录的表。

这使我们想到如下两条数据定义：

> *File*（文件）是符号。

> *directory*（目录，简称 *dir*）是下列三者之一：
> 1. **empty**。
> 2. (**cons** *f d*)，其中 *f* 是 *file*，*d* 是 *dir*。
> 3. (**cons** *d1 d2*)，其中 *d1* 和 *d2* 是 *dir*。

第一个数据定义说明文件由名字代表。第二个数据定义描述了目录是如何通过逐步添加文件和目录构造得到的。

仔细观察第二个数据定义，可以发现目录类型就是第 14.3 节中的网页类型。因此，我们可以重用网页处理函数的模板来处理目录树。如果我们要写一个函数，它读入一个目录（树）并计算其中所包含的文件总数，那么这个函数就是计算网页（树）中单词总数的函数。

习题

习题 16.2.1　按照模型一，把图 16.1 中的文件系统转化为 Scheme 表示。

习题 16.2.2　开发函数 *how-many*，该函数读入一个 *dir*，返回该 *dir* 树中的文件数。

模型二：虽然我们很熟悉第一个数据定义，而且它用起来也很方便，但是它隐藏了目录的本质。具体来说，它隐藏了这样一个事实，即目录并不只是文件和目录的集合，它还有一些有趣的属性。要建立一个更翔实的目录模型，我们必须引入一个结构体，它收集目录的所有相关属性。最简单的结构体定义如下：

```
(define-struct dir (name content))
```

它表明目录有名字，有内容。如果需要的话，我们现在还可以加上其他的属性。

新的定义的目的是表明目录有两个属性：名字和内容，名字是一个符号，而内容是文件和目录的表。这从而表明了如下两个数据定义：

> *directory*（目录，简称 *dir*）是结构体：
>
> **(make-dir** *n c***)**
>
> 其中 *n* 是符号，*c* 是文件和目录的表。

> *list-of-files-and-directories*（文件和目录的表，简称 *LOFD*）是下列三者之一：
>
> 1. **empty**。
> 2. **(cons** *f d***)**，其中 *f* 是文件，*d* 是 *LOFD*。
> 3. **(cons** *d1 d2***)**，其中 *d1* 是 *dir*，*d2* 是 *LOFD*。

因为 *dir* 的数据定义引用了 *LOFD* 的定义，而 *LOFD* 的定义又反过来引用了 *dir* 的数据定义，所以它们是相互引用的定义，必须同时引入。

粗略地来说，这两个定义的相互关系类似于第 15.1 节中 *parent* 和 *list-of-children* 的关系。这就说明第 15.2 节中的设计诀窍可以直接应用于 *dir* 和 *LOFD*。更具体地说，要设计一个处理 *dir* 的函数，我们必须并行地开发 *dir* 处理函数和 *LOFD* 处理函数的模板。

习题

习题 16.2.3　说明如何建立一个模型，其中的目录还有另外两个属性：大小和系统属性。前者测量目录本身（不是它的文件和子目录）用去了多少空间；后者表明该目录是不是操作系统所支持的。

习题 16.2.4　按照模型二，把图 16.1 中的文件系统转化为 Scheme 表示。

习题 16.2.5　开发函数 *how-many*，该函数读入一个（依照模型二的）*dir*，返回该 *dir* 树中的文件数。

模型三：第二个数据定义改进了第一个数据定义，引入了目录的属性，文件也有属性，要建立文件属性的模型，我们还是一样处理。首先，我们定义文件的结构体：

```
(define-struct file (name size content))
```

接着给出数据定义：

> *file*（文件）是结构体：
>
> **(make-file** *n s x***)**
>
> 其中 *n* 是符号，*s* 是数，而 *x* 是某个 Scheme 值。

现在，我们把文件的 *content* 字段看作一个设置为 empty 的字段。以后，我们会讨论如何存取文件中的数据。

最后，我们来把 *dir* 的 *content* 字段分成两个部分：一个部分是文件表，另一个是子目录表。文件表的数据定义很简单，它只依赖于 *file* 的定义：

> *list-of-files*（文件表）是下列二者之一：
>
> 1. **empty**。
> 2. **(cons** *s lof***)**，其中 *s* 是 *file*，而 *lof* 是文件表。

与之相反，*dir* 的数据定义及其子目录表的数据定义是相互引用的，因此，它们必须同时引入。当然，我们先需要 *dir* 的结构体定义，它有一个文件表字段和一个子目录表字段：

```
(define-struct dir (name dirs files))
```

它们的数据定义是：

> *dir*（目录）是结构体：
>
> **(make-dir n ds fs)**
>
> 其中 *n* 是符号，*ds* 是目录表，*fs* 是文件表。

> *list-of-directories*（目录表）是下列二者之一：
>
> 1. **empty**。
> 2. **(cons** *s lod***)**，其中 *s* 是 *dir*，*lod* 是目录表。

这第三个目录层次的模型（数据表示法）抓住了文件系统的本质，至少是用户一般可以观察到的本质。不过，它有两个结构体定义，四个数据定义，比第一个模型复杂得多。但是，从第一个模型的简单表示法开始，通过一步一步地改进，我们理解了如何处理这种复杂类型的组织。现在，我们的任务是，使用第 15.2 节中的设计诀窍来开发处理这个数据定义集合的函数，不然的话，我们就完全没有办法来理解这种定义。

16.3　改进函数和程序

下列练习题的目标是，使用我们的第三个模型，也就是最精确的模型，开发一些处理目录和文件系统的常用函数。虽然这些函数处理的是基于 Scheme 表示法的文件和目录，但是它可以帮助我们很好地想象真实世界中的程序是怎样运作的。

习题

习题 16.3.1　把图 16.1 中的文件系统转化为 Scheme 表示。记住用 empty 作为文件的内容。

为了使习题更为现实，DrScheme 支持教学软件包 dir.ss。该教学软件包引入了两个必需的结构体定义以及一个函数，该函数可以按照我们的第三种模型建立目录的表示：

```
;; create-dir : string -> dir
```

```
;; 建立 a-path 所指定的目录的表示:
;; 1. Windows: (create-dir "c:\\windows")
;; 2. Mac: (create-dir "My Disk:")
;; 3. Unix: (create-dir "/home/scheme/")
(define (create-dir a-path) ...)
```

使用这个函数，按照真实计算机的目录，建立一些或大或小的例子。**警告**：对于大型的目录树，DrScheme 可能需要大量的时间来建立表示。先使用 *create-dir* 来建立小型的目录。不要定义你自己的 *dir* 结构体。

习题 16.3.2 开发函数 *how-many*，该函数读入一个（依照模型三的）*dir*，返回该 *dir* 树中的文件数。用习题 16.3.1 建立的目录测试这个函数。为什么我们确信这个函数能返回正确的结果？

习题 16.3.3 开发函数 *du-dir*，该函数读入一个目录，计算整个目录树中所有文件的总长度。这个函数是对磁盘使用表的近似，因为它假设目录并不需要存储空间。

改进这个函数，近似计算子目录的长度。我们可以假设，在 *dir* 结构体中，存储 *content* 字段中的一个文件或者一个目录需要 1 个存储单元。

习题 16.3.4 开发函数 *find?*，该函数读入一个 *dir* 和一个文件名，判断在该目录树中有没有出现这个名字的文件。

挑战：开发函数 *find*。这个函数读入目录 *d* 和文件名 *f*。如果(*find? d f*)为真，该函数返回一条到达该文件的路径；否则，它返回 false。路径是目录名的表，表中的第一个目录是给定的目录；最后一个目录是这样一个子目录它的 *files* 表包含文件 *f*。例如：

```
(find TS 'part3)
;; 期望值:
(list 'TS 'Text)

(find TS 'read!)
;; 期望值:
(list 'TS)
```

假设 *TS* 被定义为图 16.1 中的目录。

在图 16.1 中，*find* 应该找到哪个 read!文件？一般化这个函数，使得它返回一个路径表，如果给定的文件名出现了多次的话。每一条路径应该到达不同的文件，而且应该包含到达每一个这样的文件的路径。



第 17 章

处理两种复杂数据片段

有时候，一个函数会读入两个参数，它们分别属于两个非平凡的数据类型。在某些情况下，其中的一个参数应当被当作原子值来处理；精确的函数用途说明一般可以阐明这一点。在其他情况下，这两个参数必须被一致地处理。最后，在某些特殊的情况下，这样的函数必须考虑所有可能的情形，并相应处理参数。本章通过例子说明了这三种情况，并针对最后一种情况给出扩充的设计诀窍。本章的最后一节讨论了复合数据的相等性，以及它与测试的关系；这一点对使用函数来进行自动测试来说是必不可少的。

17.1 同时处理两个表：第一种情况

考虑如下的合约、用途说明和头部：

```
;; replace-eol-with : list-of-numbers list-of-numbers -> list-of-numbers
;; 通过把 alon1 中的 empty 替换成 alon2，建立一个新的表
(define (replace-eol-with alon1 alon2) ...)
```

这个合约表明该函数读入两个表，而以前我们还没有遇到过这种情形。我们来看看设计诀窍在这种情况下是怎样工作的。

第一步，我们构造例子。假设第一个输入是 empty。此时 *replace-eol-with* 应该返回第二个参数，无论它是什么：

```
  (replace-eol-with empty L)
= L
```

在这个等式中，*L* 代表任意一个数表。现在假设第一个参数不是 empty。此时用途说明要求我们把 *alon1* 末尾的 empty 替换成 *alon2*：

```
(replace-eol-with (cons 1 empty) L)
;; 期望值：
(cons 1 L)

(replace-eol-with (cons 2 (cons 1 empty)) L)
;; 期望值：
(cons 2 (cons 1 L))

(replace-eol-with (cons 2 (cons 11 (cons 1 empty))) L)
;; 期望值：
```

```
(cons 2 (cons 11 (cons 1 L)))
```

同样，在这些例子中，*L* 代表任意的数表。

例子说明，第二个参数是什么并不重要——只要它是一个表就可以了；否则，用这第二个参数替代 empty 就没有意义了。这表明该函数的模板应该是一个表处理函数，它处理第一个参数：

```
(define (replace-eol-with alon1 alon2)
  (cond
    ((empty? alon1) ...)
    (else ... (first alon1) ... (replace-eol-with (rest alon1) alon2) ... )))
```

第二个参数被当作一个原子数据来处理。

我们按照设计诀窍和例子来填写模板中的空缺。如果 *alon1* 是 empty，按照例子，*replace-eol-with* 返回 *alon2*。对于第二个 cond 子句，如果 *alon1* 不是 empty，我们必须检查下列可用的表达式：

1. (**first** *alon1*)计算出表中第一个元素。
2. (*replace-eol-with* (**rest** *alon1*) *alon2*)把(**rest** *alon1*)中的 **empty** 替换成 *alon2*。

为了更好地理解它们，考虑一个例子：

```
(replace-eol-with (cons 2 (cons 11 (cons 1 empty))) L)
;; 期望值:
(cons 2 (cons 11 (cons 1 L)))
```

这里，(first *alon1*)是 2，(rest *alon1*)是(cons 11 (cons 1 empty))，而(*replace-eol-with* (rest *alon1*) *alon2*)是(cons 11 (cons 1 *alon2*))。我们可以用 cons 把 2 和后者连接起来，从而得到所需的结果。更一般地说，

(cons (first *alon1*) (*replace-eol-with* (rest *alon1*) *alon2*))

就是第二个 cond 子句的答案。图 17.1 给出了完整的定义。

```
;; replace-eol-with : list-of-numbers list-of-numbers ->list-of-numbers
;; 通过把 alon1 中的 empty 替换成 alon2，建立一个新的表
(define (replace-eol-with alon1 alon2)
  (cond
    ((empty? alon1) alon2)
    (else (cons (first alon1) (replace-eol-with (rest alon1) alon2)))))
```

图 17.1 *replace-eol-with* 的完整定义

习题

习题 17.1.1 一些习题曾用到过 Scheme 的操作 append，该操作读入三个表，并将它们的元素并列：

```
(append (list 'a) (list 'b 'c) (list 'd 'e 'f))
;; 期望值:
(list 'a 'b 'c 'd 'e 'f)
```

使用 *replace-eol-with* 来定义 *our-append*，它的行为类似于 Scheme 的 append。

习题 17.1.2 开发 *cross*，该函数读入一个符号表和一个数表，返回所有可能的符号-数对。

例如：

```
(cross '(a b c) ' (1 2))
;; 期望值:
(list .(list 'a 1). (list 'a 2) .(list 'b 1). (list 'b 2). (list 'c 1). (list'c2))
```

17.2 同时处理两个表：第二种情况

在第 10.1 节中，我们开发了函数 *hours->wages*，用来计算周工资。这个函数读入一个数表，即每周的工作时间，返回周工资的表。该函数基于一个简单的假设：所有员工获得的单位工资都是一样的。不过，即使在小公司里，不同的职员也会有不同的工资等级。一般来说，公司的会计人员会保存两种信息的集合：一种永久信息，除了记录其他的员工信息外，还记录每个员工的单位工资，另一种临时信息，记录在过去的一周中，每位员工工作了多少时间。

这个修改后的问题描述意味着该函数需要读入两个表。为了简化问题，我们假设这两个表都是数表，一个是单位工资的表，另一个是周工作时间的表。那么这个问题的描述就是这样的：

```
;; hours->wages : list-of-numbers list-of-numbers -> list-of-numbers
;; 通过相乘 alon1 和 alon2 中对应元素，建立一个新的表
;; 假设：这两个表的长度相等
(define (hours->wages alon1 alon2) ...)
```

我们可以把 *alon1* 当作单位工资的表，把 *alon2* 当作周工作时间的表。要得到周工资的表，必须把两个输入表中相应的元素乘起来。

来看一些例子：

```
(hours->wages empty empty)
;; 期望值:
empty

(hours->wages (cons 5.65 empty) (cons 40 empty))
;; 期望值:
(cons 226.0 empty)

(hours->wages (cons 5.65 (cons 8.75 empty))
              (cons 40.0 (cons 30.0 empty)))
;; 期望值:
(cons 226.0 (cons 262.5 empty))
```

对所有这三个例子来说，函数都被作用于两个等长的表。正如用途说明的结尾所说，函数假设输入的两个表长度相等，而且，事实上，如果这个条件不成立，使用这个函数是没有任何意义的。

在开发模板的过程中，我们可以利用这个输入条件。更具体地说，这个条件表明，当且仅当(empty? *alon1*)成立时，(empty? *alon2*)才成立；另外，当且仅当(cons? *alon1*)成立时，(cons? *alon2*)才成立。换一种说法，这个条件简化了模板中 cond 结构的设计，因为条件表明该模板类似于普通的表处理函数：

```
(define (hours->wages alon1 alon2)
  (cond
    ((empty? alon1) ...)
    (else ... )))
```

在第一个 cond 子句中，*alon1* 和 *alon2* 和都是 empty。因此，这时不需要任何的选择器表达式。在第二个子句中，*alon1* 和 *alon2* 都是 cons 构建的表，这意味着我们需要四个选择器表达式：

```
(define (hours->wages alon1 alon2)
  (cond
    ((empty? alon1) ...)
    (else
      ... (first alon1) ... (first alon2) ...
      ... (rest alon1) ... (rest alon2) ... )))
```

最后，因为后两个表的长度是相等的，所以它们就自然成为了 *hours->wages* 自然递归的候选参数：

```
(define (hours->wages alon1 alon2)
  (cond
    ((empty? alon1) ...)
    (else
      ... (first alon1) ... (first alon2) ...
      ... (hours->wages (rest alon1) (rest alon2)) ... )))
```

这个模板唯一不寻常的地方是，递归调用由两个表达式组成，这两个表达式分别是两个参数的选择器表达式。但是，正如我们所看到的，由于假设 *alon1* 和 *alon2* 的长度相等，这种观点很容易解释。

要由此定义函数，我们遵循设计诀窍进行。第一个例子表明，对于第一个 cond 子句来说，答案是 empty。在第二个子句中，我们有三个可以使用的值：

1. **(first** *alon1*)计算出单位工资表的第一个元素；
2. **(first** *alon2*)计算出工作时间表的第一个元素；
3. (*hours->wages* (**rest** *alon1*) (rest *alon2*))计算 *alon1* 和 *alon2* 的其余部分的周工资表。

只需把这些值结合起来就可以得到最终的结果。更明确地说，按照用途说明，我们必须计算第一个员工的周工资，并用 cons 由这个工资和其他的工资构建一个表，这表明第二个 cond 子句的答案如下：

```
(cons (weekly-wage (first alon1) (first alon2))
      (hours->wages (rest alon1) (rest alon2)))
```

辅助函数 *weekly-wage* 读入两个表的第一个元素，计算相应的周工资。图 17.2 给出了完整的定义。

```
;;hours->wages :list-of-numbers list-of-numbers->list-of-numbers
;; 通过相乘 alon1 和 alon2 中对应的元素，建立一个新的表
;; 假设：这两个表的长度相等
(define (hours->wages alon1 alon2)
  (cond
    ((empty? alon1) empty)
    (else (cons (weekly-wage (first alon1) (first alon2))
                (hours->wages (rest alon1) (rest alon2))))))

;; weekly-wage : number number -> number
;; 由 pay-rate（单位工资）和 hours-worked（工作时间）计算周工资
(define (weekly-wage pay-rate hours-worked)
  (* pay-rate hours-worked))
```

图 17.2　*hours→wage* 的完整定义

习题

习题 17.2.1 在真实世界中，*hours->wages* 读入员工结构体的表和工作结构体的表。员工结构体包含员工的名字、社会保险号码和单位工资（每小时工资）。工作结构体包含员工的名字和他一周中的工作时间。函数的返回值是一个结构体的表，该结构体中包含员工的名字和周工资。

修改图 17.2 中的函数，让它处理这些数据类型。给出必要的结构体定义和数据定义。使用设计诀窍来指导修改过程。

习题 17.2.2 开发函数 *zip*，该函数把人名表和电话号码表结合成电话记录表。假定其结构体定义如下：

```
(define-struct phone-record (name number))
```

电话记录是由(make-phone-record *s n*)构造的，其中 *s* 是符号，*n* 是数。假设输入的表是等长的。尽可能简化你的定义。

17.3 同时处理两个表：第三种情况

这是第三种问题描述，以函数的合约、用途说明以及头部的形式给出：

```
;; list-pick : list-of-symbols N[>= 1] -> symbol
;; 求出 alos 中的第 n 个符号，从 1 开始计数;
;; 如果没有第 n 个元素，发出一个错误信息
(define (list-pick alos n) ...)
```

换一种说法，这个问题是要开发一个函数，它读入一个自然数和一个符号表。这两者都属于具有复杂数据定义的类型，不过，不同于前两个问题，这两种类型并不相同。图 17.3 回忆了这两个定义。

数据定义：

natural number(自然数) [>= 1]（N[>= 1]）是下列二者之一：
1. **1**。
2. **(add1** *n***)**，如果 *n* 是 **N[>= 1]**。

list-of-symbols（符号表）是下列二者之一：
1. 空表 **empty**。
2. **(cons** *s lof***)**，其中 *s* 是符号，*lof* 是符号表。

图 17.3 *list-pick* 的数据定义

因为这个问题并不标准，所以应该确保例子能覆盖所有的主要情况。一般从定义的每个子句中随意选择一个元素，或者从每个基本数据形式中随意选择一个元素，从而保证例子能覆盖所有的主要情况。在这个例子中，这种过程意味着我们必须从 *list-of-symbols* 中选择两个元素，从 N[>=1]中也选择两个元素。但是，每个参数各两种选择说明总共有四种例子；毕竟，在这两个参数之间没有直接的联系，合约中也没有任何关于参数的限制：

```
(list-pick empty 1)
;; 期望的行为:
(error 'list-pick "...")
```

```
(list-pick (cons 'a empty) 1)
;; 期望值:
'a

(list-pick empty 3)
;; 期望的行为:
(error 'list-pick "...")

(list-pick (cons 'a empty) 3)
;; 期望的行为:
(error 'list-pick "...")
```

这四个例子只有一个返回符号；在其他的情况下，我们会看到错误信息，表明表中没有足够的元素。

对例子的讨论表明，实际上共有四种独立的情况，我们必须在函数设计时处理这四种可能的情况。可以用表格的形式列出所有必需的条件，从而得到这四种情况：

	(empty? *alos***)**	**(cons?** *alos***)**
(= *n* **1)**		
(> *n* **1)**		

在该表格中，第一行列出了 *list-pick* 必须对表参数进行的判断；第一列列出了 *list-pick* 必须对自然数参数进行的判断。因此，我们得到了四个方格，每个方格代表一种情况，即该格所在的行和列中的条件都成立，可以用 and 表达式表示这种情况：

	(empty? *alos***)**	**(cons? alos)**
(= *n* **1)**	**(and (=** *n* **1)** **(empty?** *alos***))**	**(and (=** *n* **1)** **(cons?** *alos***))**
(> *n* **1)**	**(and (>** *n* **1)** **(empty?** *alos***))**	**(and (>** *n* **1)** **(cons?** *alos***))**

显然，这四个复合条件中正好会有一个成立。

基于这些情况分析，我们现在可以设计模板的第一个部分，即条件表达式：

```
(define (list-pick alos n)
  (cond
    [(and (= n 1) (empty? alos)) ...]
    [(and (> n 1) (empty? alos)) ...]
    [(and (= n 1) (cons? alos)) ...]
    [(and (> n 1) (cons? alos)) ...]))
```

这个 cond 表达式询问所有四种可能的问题。接下来我们必须在每个 cond 子句中添上所有可能的选择器表达式：

```
(define (list-pick alos n)
  (cond
    [(and (= n 1) (empty? alos))
     ...]
    [(and (> n 1) (empty? alos))
     ... (sub1 n) ...]
```

```
      [(and (= n 1) (cons? alos))
       ... (first alos) ... (rest alos)...]
      [(and (> n 1) (cons? alos))
       ... (sub1 n) ... (first alos) ... (rest alos) ...]))
```

对于自然数 *n* 来说，模板必须至少包含一个选择器表达式，该表达式求出 *n* 的前趋。对于 *alos* 来说，模板可能就要包含两个选择器表达式。如果(= *n* 1)或者(empty? *alos*)成立，那么这两个参数中就至少有一个是原子值，我们就没有必要对它使用选择器表达式了。

遵循构建模板的最后一个步骤，当选择器表达式的返回值与输入属于相同的类型时，我们用递归注释模板。在 *list-pick* 的模板中，这一条仅对最后一个 cond 子句有效，因为该子句既包含了 N[>=1]的选择器表达式，又包含了 *list-of-symbols* 的选择器表达式。其他的所有子句最多只包含一个相关的选择器表达式。不过，我们还不清楚自然递归的形式是怎样的。如果我们不考虑函数的用途，而只是按照模板构建步骤的要求来做，就有三种可能的递归形式：

```
1. (list-pick (rest alos) (sub1 n))
2. (list-pick alos (sub1 n))
3. (list-pick (rest alos) n)
```

既然我们不知道需要其中的哪一个，或者是不是需要所有这三个，就先进入下一个开发阶段。

遵照设计诀窍，我们来分析模板中的每一个 cond 子句，并确定正确的答案是什么：

1．如果(and (= *n* 1) (empty? *alos*))成立， *list-pick* 被要求从一个空表中选取出第一个元素，而这是不可能的，这时的答案必然是错误。

2．如果(and (> *n* 1) (empty? *alos*))成立，*list-pick* 还是被要求从一个空表中选取出一个元素，这时的答案还是错误。

3．如果(and (= *n* 1) (cons? *alos*))成立，那么 *list-pick* 就应该从某个表返回第一个元素。选择器表达式(first *alos*)提示怎样获得这个项，它就是答案。

4．对于最后一个子句，如果(and (> *n* 1) (cons? *alos*))成立，我们必须分析选择器表达式会计算出什么：

a．(first *alos*)从符号表中选出第一个元素；

b．(rest *alos*)是表的其余部分；

c．(sub1 *n*)是比表的给定下标小一的下标。

我们通过一个例子来说明这些表达式的含义。假设 *list-pick* 被作用于(cons 'a (cons 'b empty))和 2：

```
(list-pick (cons 'a (cons 'b empty)) 2)
```

答案必定是'b。(first *alos*)是'a，(sub1 *n*)是 1。三个自然递归分别会计算出：

a．(*list-pick* (cons 'b empty) 1)返回'b，也就是我们所需的答案；

b．(*list-pick* (cons 'a (cons 'b empty)) 1)计算出'a，也就是一个符号，但它不是原来问题正确的解；

c．(*list-pick* (cons 'b empty) 2)发出一个错误信息，因为下标比表的长度大。

这表明，我们可以用(*list-pick* (rest *alos*) (sub1 *n*))作为最后一个 cond 子句的答案。但是，基于例子的推理往往是靠不住的，所以我们应该设法理解为什么这个表达式总能正确工作。

回忆一下，按照用途说明，

```
(list-pick (rest alos) (sub1 n))
```

选出(rest *alos*)中第（*n*－1）个元素。换句话说，在这第二个调用中，我们把下标减 1，把表缩短一个项，然后寻找元素。显然，假设 *alos* 和 *n* 是“复合的”值，那么这第二个调用总能返回与第一个调用相同的答案。这证明了我们对最后一个子句的选择是正确的。

List-pick 的完整定义见图 17.4

```
;; list-pick : list-of-symbols N[>=1]-> symbol
;; 求出 alos 中的第 n 个符号，从 1 开始计数；
;; 如果没有第 n 个元素，发出一个错误信息
(define (list-pick alos n)
  (cond
    [(and (= n 1) (empty? alos)) (error 'list-pick "list too short")]
    [(and (> n 1) (empty? alos)) (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (sub1 n))]))
```

图 17.4　*list-pick* 的完整定义

习题

习题 17.3.1　开发 *list-pick0*，该函数从表中选出一个元素，类似于 *list-pick*，但是从 **0** 开始计数。例如：

```
(symbol=? (list-pick0 (list 'a 'b 'c 'd) 3)
        'd)
(list-pick0 (list 'a 'b 'c 'd) 4)
;; 预期的行为：
(error 'list-pick0 "the list is too short")
```

17.4　函数的简化

图 17.4 中的 *list-pick* 函数过于复杂了，它的第一和第二个子句都返回相同的答案：错误。换句话说，如果

```
(and (= n 1) (empty? alos))
```

或者

```
(and (> n 1) (empty? alos))
```

中有一个计算出 **true**，函数的答案就是错误。我们可以把这个观察结论转变为一个更简单的 **cond** 表达式：

```
(define (list-pick alos n)
  (cond
    [(or (and (= n 1) (empty? alos))
         (and (> n 1) (empty? alos))) (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (sub1 n))]))
```

这个新的表达式是由我们的观察结果直接翻译成 Scheme 语言所得的。

为了进一步简化这个函数，我们需要了解一个布尔代数法则：

```
  (or (and condition1 a-condition)
      (and condition2 a-condition))
= (and (or condition1 condition2)
       a-condition)
```

这个法则被称作德摩根分配律，把它应用于我们的函数，就可以得到如下结果：

```
(define (list-pick alos n)
  (cond
    [(and (or (= n 1) (> n 1))
          (empty? alos)) (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (sub1 n))]))
```

现在，考虑条件的第一个部分：**(or (= *n* 1) (> *n* 1))**。因为 *n* 属于 **N[>= 1]**，所以这个条件总是为真。但是，如果我们用 **true** 代替它，就会得到：

```
(and true
     (empty? alos))
```

而这显然等价于**(empty?** *alos***)**。换句话说，这个函数可以写成

```
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(and (= n 1) (cons? alos)) (first alos)]
    [(and (> n 1) (cons? alos)) (list-pick (rest alos) (sub1 n))]))
```

比起图 17.4 中的定义，这已经得到了显著的简化。

尽管如此，我们还可以做得更好。在刚才的 *list-pick* 版本中，第一个条件选出所有那些 *alos* 为空的情况。因此，在后两个子句中，(cons? *alos*)总会计算出 true。如果我们用 true 来代替这个条件，并简化 and 表达式，就可以得到最为简单的 *list-pick* 版本，图 17.5 给出了这个函数。虽然这最后一个函数要比原来的简单，但是很重要的一点是，这两个函数都是用系统化的方法开发的，我们可以相信它们。如果试图直接开发函数的简单版本，迟早会在考虑某种情况时出错，从而得到有缺陷的函数。

```
;; list-pick : list-of-symbols N[>= 1] -> symbol
;; 求出 alos 中的第 n 个符号，从 1 开始计数；
;; 如果没有第 n 个元素，发出一个错误信息
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(= n 1) (first alos)]
    [(> n 1) (list-pick (rest alos) (sub1 n))]))
```

图 17.5 *list-pick* 的简化定义

习题

习题 17.4.1 遵照第 17.2 节中的方法，开发函数 *replace-eol-with*，然后用系统化方法对它进行简化。

习题 17.4.2 简化习题 17.3.1 中的函数 *list-pick0*，或者解释为什么它不能被简化。

17.5 设计读入两个复杂输入的函数

有时候，我们会遇到这样的问题，它要求函数读入两种复杂类型的输入。其中最有趣的情况是，这

两个输入都是不定长的。正如在前三节中所看到的，我们会用三种不同的方法来处理这种函数。

解决这个问题的正确方法是遵循一般的设计诀窍。具体来说，我们必须进行数据分析，定义相关的数据类型，然后给出函数的合约和用途说明。在继续进行设计之前，应该确定正在处理的情况是下列三种中的哪一个：

1．某些情况下，有一个参数起支配作用。在这个函数中，我们可以把另一个参数看作原子数据。

2．在其他一些情况下，两个参数是同步的，它们必定涉及到同一种类型的值，而且它们的结构也是相同的。例如，如果输入是两个表，它们必然是等长的。如果输入是两个网页，它们必然是等长的，而且，如果一个网页包含嵌入网页，那么另一个网页也包含嵌入网页。如果判断出这两个参数具备这种相同的状态，我们就可以从它们之中选择一个，围绕它来组织函数。

3．最后，在少数情况下，两个参数之间可能没有什么明显的关联。碰到这种情况，在挑选例子和设计模板之前，必须分析所有可能的情况。

对前两种情况来说，可使用现有的设计诀窍，最后一种情况需要我们进行一些特别的考虑。

在确定某个函数属于这第三种情况之后，在开发函数的例子和模板之前，我们可以使用一个两维的表格，如 *list-pick* 的表格：

	alos		
		(empty? *alos*)	(cons? *alos*)
n	(= *n* 1)		
	(> *n* 1)		

第一行列举出第一个参数的所有子类型，第一列列举出第二个参数的所有子类型。

这个表格指导我们开发函数的例子以及函数的模板。对于所需的例子来说，它们必须覆盖所有可能的情况。也就是说，对于表格中的每一个方格，至少要有一个例子。

对应于每个方格，模板必须包含一个 cond 子句。反过来说，每个 cond 子句至少要包含两个参数所有可能的选择器表达式。如果某个参数是原子的，它就不需要选择器表达式。最后，我们可能会得到多个自然递归表达式。对于 *list-pick* 来说，我们就得到了三个自然递归。一般来说，所有可能的选择器表达式组合都可以形成自然递归。因为不知道哪个自然递归是必需的，而哪个又不是必需的，所以我们把它们都写下来，然后在真正定义函数时再从中选择。

总而言之，多参数函数的设计只需稍加修改原来设计诀窍即可。关键的思想是把数据定义转化成表格，用表格说明所有需要处理的、可能的组合。函数的例子与模板的开发应尽可能利用这个表格。就像以前一样，填写模板中的空缺需要练习。

17.6　处理两个复杂输入的练习

习题

习题 17.6.1　开发函数 *merge*，该函数读入两个升序排列的数表，返回一个升序排列的数表，该表中包含（且仅包含）两个输入表中所有的数，某个数在输出表中出现的次数应该与它在两个输入表中出现的总数相同。

例如：

```
(merge (list 1 3 5 7 9) (list 0 2 4 6 8))
;; 期望值:
(list 0 1 2 3 4 5 6 7 8 9)
```

```
(merge (list 1 8 8 11 12) (list 2 3 4 8 13 14))
;; 期望值:
(list 1 2 3 4 8 8 8 11 12 13 14)
```

习题 17.6.2 本习题的目标是，开发第 6.7 节中刽子手游戏的一个新版本，它能处理任意长的单词。用表的形式给出代表任意长单词的数据定义。字母（*letter*）是用'a 到'z 以及'_的符号表示的。

开发函数 *reveal-list*，该函数读入三个参数：

1. 所选单词（*chosen*），也就是我们要猜的单词；
2. 状态单词（*status*），表示已经猜出的部分单词；
3. 一个字母（*guess*），也就是我们当前的猜测。

函数返回一个新的状态单词，也就是一个包含普通字母以及'_的单词。为了得出新状态单词中的字段，需要比较猜测字母以及（原）状态单词和所选单词的每一对字母：

1. 如果猜测等于所选单词中的某个字母，新状态单词中相应的字母就是这个猜测；
2. 否则，新状态单词就是（原）状态单词中相应的字母。

用下列例子测试这个函数：

```
1. (reveal-list (list 't 'e 'a) (list '_ 'e '_) 'u)
2. (reveal-list (list 'a 'l 'e) (list 'a '_ '_) 'e)
3. (reveal-list (list 'a 'l 'l) (list '_ '_ '_) 'l)
```

先求出返回值应该是什么。

用教学软件包 hangman.ss 以及（习题 6.7.1 中的）函数 *draw-next-part* 和 *reveal-list* 来试玩游戏。计算表达式：

```
(hangman-list reveal-list draw-next-part)
```

函数 *hangman-list* 随机选择一个单词，然后弹出一个窗口，窗口中有一个字母的选择菜单。选择一个字母，然后单击 Check 按钮，看看你作的猜测对不对。享受游戏的乐趣吧！

习题 17.6.3 在某个工厂里，员工早上到达以及晚上离开时都要打卡（用计时钟在考勤卡上打印上下班时间）。现代化的电子考勤卡记录了员工的号码和工作时间。另外，员工记录包含员工的名字、号码和单位工资。

开发函数 *hours->wages2*，该函数读入一个员工记录表和一个（电子）考勤卡表。函数对员工记录和考勤卡中的员工号码进行匹配，计算每位员工的周工资。如果某条员工记录和（电子）考勤卡不能匹配，或者无法匹配，函数就发出一个相应的错误信息并停止运行。假设每一位员工最多只有一张考勤卡，每个员工号码也最多只有一张考勤卡。

提示：会计人员一般会先按照员工号码对这两个表排序。

习题 17.6.4 线性组合是线性元素之和，而线性元素是变量和数的积，这里的数被称作系数。下面是一些例子：

$$5 \cdot x$$
$$5 \cdot x + 17 \cdot y$$
$$5 \cdot x + 17 \cdot y + 3 \cdot z$$

在这三个例子中，x 的系数是 5，y 的系数是 17，z 的系数是 3。

如果给定了变量的值，我们就可以求出多项式的值。例如，如果 $x=10$，$5 \cdot x$ 的值就是 50；如果 $x=10$ 而 $y=1$，$5 \cdot x+17 \cdot y$ 的值就是 67；如果 $x=10$，$y=1$ 而 $z=2$，$5 \cdot x+17 \cdot y+3 \cdot z$ 的值就是 73。

过去人们用函数来计算线性组合的值。线性组合的另一种表示法是系数表示法，上述三个线性组合可以表示为：

```
(list 5)
(list 5 17)
```

```
(list 5 17 3)
```

这个表示法假设我们总是按照某个固定的顺序排列变量。

开发函数 *value*，该函数读入一个多项式的表示以及一个数表，这两个表是等长的。函数返回多项式关于这些值的值。

习题 17.6.5　路易斯、简、劳拉、达纳和玛丽是姐妹，她们积攒了一些钱，用来购买圣诞节礼物（每个人买一份礼物，送给另一个人）。所以，她们决定进行抓阄，为每个人分配一个接受其礼物的人。因为简是计算机程序员，所以由她来写一个程序，公平地执行这次抓阄。当然这个程序不能把任何一个人的礼物分配给她自己。

下面是 *gift-pick* 的定义，该函数读入一个不同名字（符号）的表，随机地选择该表的一个排列，使这个表与原来的表在每一个位置上都不相等：

```
;; gift-pick: list-of-names -> list-of-names
;; 为 names “随机地”选择一个不相同的排列
(define (gift-pick names)
  (random-pick
    (non-same names (arrangements names))))
```

回忆一下，***arrangements***（参见习题 12.4.2）读入一个符号表，返回表中元素所有排列组成的表。

开发辅助函数：

1. *random-pick : list-of-list-of-names -> list-of-names*，这个函数读入一个表，从中随机地选择一个作为返回值；

2. *non-same: list-of-names list-of-list-of-names->list-of-list-of-names*，这个函数读入名字的表 *L* 和一个排列的表，返回在所有位置上都和 *L* 不同的排列的表。

如果对两个排列调用相同次数的 rest 操作，再调用一次 first 操作，可以提取出相同的名字，那么这两个排列就在某个位置上相等。例如，(list 'a 'b 'c)和(list 'c 'a 'b)并不在某个位置上相等，而(list 'a 'b 'c)和(list 'c 'b 'a) 就在某个位置上相等。通过对这两个表调用一次 rest，再调用一次 first，就可以证明这一点。

遵照适当的诀窍，仔细地进行函数设计。

提示：回忆一下，(random *n*)选出一个在 0 和 *n*-1 之间的随机数（比较习题 11.3.1）。

习题 17.6.6　开发函数 *DNAprefix*，该函数读入两个参数，这两个参数都是符号表（在 DNA 中，只存在'a、'c、'g 和't 四种符号，但是这里可以忽略这个问题）。前一个表被称为模式（pattern），后一个表是搜索字符串（search-string）。如果模式是搜索字符串的前缀，函数就返回 true；在其他所有情况下，函数都返回 false。

例子：

```
(DNAprefix (list 'a 't) (list 'a 't 'c))
(not (DNAprefix (list 'a 't) (list 'a)))
(DNAprefix (list 'a 't) (list 'a 't))
(not (DNAprefix (list 'a 'c 'g 't) (list 'a 'g)))
(not (DNAprefix (list 'a 'a 'c 'c) (list 'a 'c)))
```

如果可能的话，简化 *DNAprefix*。

修改 *DNAprefix*，使得它返回搜索字符串中位于模式之后的第一个元素，如果模式是搜索字符串（真正意义上）的前缀的话。如果这两个表并不匹配，或者如果模式比搜索字符串长，修改后的函数应该还是返回 false。类似地，如果这两个表的长度相等，返回值还是 true。

例子：

```
(symbol=? (DNAprefix (list 'a 't) (list 'a 't 'c))
```

```
        'c)
(not (DNAprefix (list 'a 't) (list 'a)))
(DNAprefix (list 'a 't) (list 'a 't))
```

可以对 *DNAprefix* 的这个变体进行简化吗？如果可以，对它进行简化。如果不行，解释为什么。

17.7 补充练习：Scheme 求值之二

这一节，我们扩展第 14.4 节中的求值程序，使它可以应付函数调用和函数定义。换句话说，这个新的求值程序模仿 DrScheme 中向 Interactions 窗口输入一个表达式，并单击 Execute 按钮后所发生的事情。为了简单起见，我们假设 Definitions 窗口中所有的函数都只读入一个参数。

习题

习题 17.7.1 扩展习题 14.4.1 中的数据定义，使其可以表示函数调用，即在某个表达式中调用用户定义的函数，例如(*f* (+ 1 1))或者(* 3 (*g* 2))等。函数调用用包含两个字段的结构体表示，前一个字段代表函数名，后一个字段代表参数。

完整的求值程序应该还可以处理函数定义。

习题 17.7.2 给出定义的结构体定义和数据定义。回忆一下，函数定义有三个基本属性：

1. 函数的名字。
2. 参数的名字。
3. 函数的主体。

这表明我们要引入一个结构体，包含三个字段，前两个字段是符号，最后一个字段代表函数的主体，也就是一个表达式。

把下列定义转化为 Scheme 值：

1. (define (*f x*) (+ 3 *x*))
2. (define (*g x*) (* 3 *x*))
3. (define (*h u*) (*f* (* 2 *u*)))
4. (define (*i v*) (+ (* *v v*) (* *v v*)))
5. (define (*k w*) (* (*h w*) (*i w*)))

构造更多例子，再把它们转化为 Scheme 值。

习题 17.7.3 开发 *evaluate-with-one-def*，该函数读入一个 Scheme 表达式（的表示）和函数定义（的表示）*P*。

习题 14.4.1 中的其余表达式可以像以前一样计算。对于变量（的表示），函数产生一个错误信息。对于函数调用 *P*，*evaluate-with-one-def* 将：

1. 计算参数；
2. 用参数的值替换函数主体中的参数；
3. 通过递归，计算新的表达式。下面是该思想的概要：[1]

```
(evaluate-with-one-def (subst ... ... ...)
                       a-fun-def)
```

对于其他所有函数调用，*evaluate-with-one-def* 产生一个错误信息。

[1] 我们会在第五部分详细讨论这种形式的递归。

习题 17.7.4　开发函数 *evaluate-with-defs*，该函数读入一个 Scheme 表达式（的表示）和函数定义（的表示）的表 *defs*。该函数模拟 DrScheme，就好像在 Interactions 窗口中计算这个真正的 Scheme 表达式，而且 Definitions 窗口包含真正的定义，函数返回 DrScheme 会返回的值。

习题 14.4.1　中其余的表达式还是和以前一样计算。对于函数 *P* 的调用，*evaluate-with-defs* 将：

1. 计算参数；
2. 在 *defs* 中查找 *P* 的定义；
3. 用参数的值代替函数主体中的参数；
4. 通过递归计算新的表达式。

类似于 DrScheme，对于表中没有的函数调用，或者对于变量（的表示），*evaluate-with-defs* 产生一个错误信息。

17.8　相等与测试

我们所设计的许多函数都返回表。测试这些函数必须比较它们的返回值和期望值，而它们都是表。手工对表进行比较相当单调乏味，而且很容易出错。我们来开发一个函数，它读入两个数表，判断它们是否相等：

```
;; list=? : list-of-numbers list-of-numbers -> boolean
;; 判断 a-list 和 another-list 是不是以相同的
;; 顺序包含相同的数
(define (list=? a-list another-list) ...)
```

这个用途说明修正了一般性的说明，提醒我们，在某些情况下，例如对购物者来说，只要两个表中包含相同的元素，而不论其顺序，它们就是相等的，但是，程序设计者应该做得更精确，要求比较的顺序也是相同的。合约和用途说明还表明，*list=?*是一个函数，它处理两个复杂值，而且对它进行研究确实相当有趣。

比较两个表意味着考察它们其中的每一个元素。这说明设计 *list=?*不可能按照第 17.1 节中 *replace-eol-with* 的方式来进行。乍一看，这两个表之间也没有任何的关联，这就表明我们应该使用修改后的设计诀窍。

我们从下面的表格开始：

	(empty? *a-list*)	(cons? *a-list*)
(empty? *another-list*)		
(cons? *another-list*)		

表格中有四个方格，这意味着我们（至少）需要四个测试，而且模板中需要有四个 cond 子句。下面是五个测试：

```
(list=? empty empty)
(not
  (list=? empty (cons 1 empty)))
(not
  (list=? (cons 1 empty) empty))
(list=? (cons 1 (cons 2 (cons 3 empty)))
        (cons 1 (cons 2 (cons 3 empty))))
(not
```

```
(list=? (cons 1 (cons 2 (cons 3 empty)))
        (cons 1 (cons 3 empty))))
```

第二和第三个测试表明 *list=?*必须用一种对称的方式处理它的参数，后两个测试说明了 *list=?*怎样返回 true 和 false。

在模板的四个 cond 子句中，有三个子句包含选择器表达式，一个子句包含自然递归：

```
(define (list=? a-list another-list)
  (cond
    [(and (empty? a-list) (empty? another-list)) ...]
    [(and (cons? a-list) (empty? another-list))
     ... (first a-list) ... (rest a-list) ...]
    [(and (empty? a-list) (cons? another-list))
    ... (first another-list) ... (rest another-list) ...]
    [(and (cons? a-list) (cons? another-list))
     ... (first a-list) ... (first another-list) ...
     ... (list=? (rest a-list) (rest another-list)) ...
     ... (list=? a-list (rest another-list)) ...
     ... (list=? (rest a-list) another-list) ...]))
```

在第四个子句中，共有三个自然递归，因为我们既可以组合两个选择器表达式，也可以组合一个参数和一个选择器表达式。

从模板到完整的函数定义之间的变化不大。仅当两个表都为空或者都是 cons 构成的情况下，它们才可能会相等。这直接表明，第一个子句的答案是 true，中间两个子句的答案是 false。在最后一个子句中，有两个数，它们分别是两个表的第一个元素，还有三个自然递归。我们必须先比较这两个数。另外，(*list=?* (rest *a-list*) (rest *another-list*))计算出这两个表的其余部分是不是相等。当且仅当这两个条件都成立时，两个表才是相等的，这表明我们必须用 and 把它们结合起来：

```
(define (list=? a-list another-list)
  (cond
    [(and (empty? a-list) (empty? another-list)) true]
    [(and (cons? a-list) (empty? another-list)) false]
    [(and (empty? a-list) (cons? another-list)) false]
    [(and (cons? a-list) (cons? another-list))
     (and (= (first a-list) (first another-list))
          (list=? (rest a-list) (rest another-list)))]))
```

另外两个自然递归没有任何作用。

我们再来观察一下这两个参数之间的联系。初次设计表明，如果两个表相等，那么两个参数就会有相同的形状。换一种说法，我们可以基于某一个参数的结构来开发函数，并按照需要检查另一个参数的结构。

第一个参数是数表，所以我们可以重用表处理函数的模板：

```
(define (list=? a-list another-list)
  (cond
    [(empty? a-list) ...]
    [(cons? a-list)
     ... (first a-list) ... (first another-list) ...
     ... (list=? (rest a-list) (rest another-list)) ...]))
```

这个模板与普通的表处理模版唯一的区别是，第二个子句还处理第二个参数，处理的方式和第一个

参数一样。这一点模仿了第 17.2 节中 *hours->wages* 的开发。

填写这个模板中的空缺就比第一次开发 *list=?*难多了。如果 *a-list* 为 empty，答案取决于 *another-list*。正如例子所示的，当且仅当 *another-list* 也是 empty 时，答案才是 true。翻译成 Scheme，这表明第一个 cond 子句的答案是(**empty?** *another-list*)。

如果 *a-list* 不为空，模板建议我们用下列表达式来计算答案：

1. (**first** *a-list*)，*a-list* 的第一个数；
2. (**first** *another-list*)，*another-list* 的第一个数；
3. (*list=?* (**rest** *a-list*) (**rest** *another-list*))，它判断表的其余部分是否相等。

根据函数的用途和例子，我们现在只需比较(first *a-list*)和(first *another-list*)，再用 and 表达式把这个结果和自然递归连接起来：

```
(and (= (first a-list) (first another-list))
     (list=? (rest a-list) (rest another-list)))
```

虽然这个步骤看上去非常简单明了，但是它是一个错误的定义。我们要求在 cond 表达式中清楚地给出所有的条件，这样做的目的是保证所有的选择器表达式都是正确的。不过，在 *list=?*的说明中，没有哪一条表明，如果 *a-list* 是 cons 结构，那么 *another-list* 也是 cons 结构。

通过使用另一个条件，我们可以解决这个问题：

```
(define (list=? a-list another-list)
  (cond
    [(empty? a-list) (empty? another-list)]
    [(cons? a-list)
     (and (cons? another-list)
          (and (= (first a-list) (first another-list))
               (list=? (rest a-list) (rest another-list))))]))
```

这个条件是(cons? *another-list*)，它表明，如果(cons? *a-list*)为真，而且(cons? *another-list*)为空，那么 *list=?*返回 false。正如例子所示，这就是所需的输出。

总而言之，*list=?*表明，有时候，我们可以使用多种设计诀窍来开发一个函数。使用不同的诀窍所得的结果是不同的，不过它们之间还是紧密相关的；事实上，我们可以证明，这两者对于相同的输入总是返回相同的计算结果。另外，第二次开发受助于第一次。

习题

习题 17.8.1　对 *list=?*的两个版本进行测试。

习题 17.8.2　简化 *list=?*的第一个版本。也就是说，融合有着相同返回值的相邻 cond 子句，用 or 表达式结合它们的条件；按照需要转化 cond 子句；在最终的版本中，最后一个子句应使用 else（作条件）。

习题 17.8.3　开发 *sym-list=?*，该函数判断两个符号表是不是相等。

习题 17.8.4　开发 *contains-same-numbers*，该函数判断两个数表是否包含相同的数，而不管它们的顺序。例如：

```
(contains-same-numbers (list 1 2 3) (list 3 2 1))
```

计算出 true。

习题 17.8.5　数、符号和布尔值类型有时候被称作原子类型[1]：

[1] 有些人把 **empty** 和字符(char)也算作原子类型。

> *atom*（原子）是下列三者之一：
> 1. 数
> 2. 布尔值
> 3. 符号

开发函数 *list-equal?*，该函数读入两个原子表，判断它们是否相等。

比较 *list=?*的两个版本，可知第二个版本比第一个更易于理解。第二个定义说，如果第二个复合值是用与第一个复合值一样的构造器构成的，而且成分也相等，那么这两个复合值就是相等的。一般来说，这种思想对其他相等函数的开发是一个很好的指导。

我们来观察简单网页的相等函数，从而证实这个判断：

```
;; web=? : web-page web-page -> boolean
;; 判断 a-wp 和 another-wp 的树形是否相同，
;; 并且以相同的顺序包含相同的符号
(define (web=? a-wp another-wp) ...)
```

回忆简单网页的数据定义：

> *Web-page*（网页，简称 *WP*）是下列三者之一：
> 1. empty。
> 2. (cons *s* *wp*)，
>   其中 *s* 是符号，*wp* 是网页。
> 3. (cons *ewp* *wp*)，
>   其中 *ewp* 和 *wp* 都是网页。

这个数据定义包含三个子句，这意味着，如果要使用修改后的设计诀窍开发 *web=?*，我们就需要研究九种情况。换一种方法，我们可以使用开发 *list=?*所获得的经验，从普通的网页模板开始设计：

```
(define (web=? a-wp another-wp)
  (cond
    [(empty? a-wp) ...]
    [(symbol? (first a-wp))
     ... (first a-wp) ... (first another-wp) ...
     ... (web=? (rest a-wp) (rest another-wp)) ...]
    [else
     ... (web=? (first a-wp) (first another-wp)) ...
     ... (web=? (rest a-wp) (rest another-wp)) ...]))
```

在第二个 **cond** 子句中，我们还是按照 *hours->wages* 和 *list=?*的例子来开发。换句话说，我们假定，如果 *another-wp* 和 *a-wp* 相等，那么它们必然有着相同的形状，所以我们可以用同样的方法处理两个网页。第三个子句也可以这样处理。

现在，在把模板改进为完整定义的过程中，我们还是必须要添加两个关于 *another-wp* 的条件，从而保证选择器表达式的正确性：

```
(define (web=? a-wp another-wp)
  (cond
```

```
[(empty? a-wp) (empty? another-wp)]
[(symbol? (first a-wp))
 (and (and (cons? another-wp) (symbol? (first another-wp)))
 (and (symbol=? (first a-wp) (first another-wp))
      (web=? (rest a-wp) (rest another-wp))))]
[else
 (and (and (cons? another-wp) (list? (first another-wp)))
 (and (web=? (first a-wp) (first another-wp))
      (web=? (rest a-wp) (rest another-wp))))]))
```

具体说来，我们必须确保在第二个和第三个子句中，*another-wp* 是 cons 构成的表，它的第一个元素分别是符号或者是表。除了这一点之外，这个函数都类似于 *list=?*，并且以相同的方式运行。

习题

习题 17.8.6　根据简单网页的数据定义，画出表格。对于表格中的九种情况，各开发（至少）一个例子。用这些例子测试 *web=?*。

习题 17.8.7　开发函数 *posn=?*，该函数读入两个二元的 *posn* 结构体，判断它们是否相等。

习题 17.8.8　开发函数 *tree=?*，该函数读入两棵二叉树，判断它们是否相等。

习题 17.8.9　考虑如下两个相互递归的数据定义：

> *Slist* 是下列两者之一：
> 1. empty
> 2. (cons *s sl*)，其中 *s* 是 *Sexpr*，*sl* 是 *Slist*。

> *Sexpr* 是下列四者之一：
> 1. 数
> 2. 布尔值
> 3. 符号
> 4. *Slist*

开发函数 *Slist=?*，该函数读入两个 *Slist*，判断它们是否相等。类似于数表，如果两个 *Slist* 在相同的位置上包含相同元素，它们就是相等的。

现在，我们已经研究了值相等的概念，可以转而研究本节的初始动机：对函数进行测试了。假设我们要测试第 17.2 节中的 *hours->wages*：

```
(hours->wages (cons 5.65 (cons 8.75 empty))
              (cons 40 (cons 30 empty)))
= (cons 226.0 (cons 262.5 empty))
```

如果我们只是把函数调用输入到 Interactions 窗口中，或者把它添加到 Definitions 窗口的底部，我们就必须通过观察来比较返回值和期望值。对于较短的表，例如上述的表，这样做是可行的；对于很长的表、深层的网页或者其他较长的复合数据，人工检查就很容易出错了。

使用类似于 *list=?*的相等函数，就不必再人工检查测试结果了。在这个例子中，只要将表达式

```
(list=?
  (hours->wages (cons 5.65 (cons 8.75 empty))
```

```
                    (cons 40 (cons 30 empty)))
  (cons 226.0 (cons 262.5 empty)))
```

添加到 Definitions 窗口的底部。现在，我们单击 Execute 按钮，只需确认所有的测试情况都返回 true，也就是在 Interactions 窗口中它们的返回值都是 true 就行了。

```
;; test-hours->wages : list-of-numbers list-of-numbers
;;list-of-numbers -> test-result
;; 对 hours->wages 进行测试
(define (test-hours->wages a-list another-list expected-result)
  (cond
    [(list=? (hours->wages a-list another-list) expected-result)
     true]
    [else
     (list "bad test result:" a-list another-list expected-result)]))
```

图 17.6 一个测试函数

事实上还可以做得更好，我们可以写出测试函数，如图 17.6 中的函数所示。*test-result* 类型由值 true 和一个表组成，该表中包含四个元素：字符串"bad test result:"和另外三个表。使用这个新的辅助函数，我们可以这样测试 *hours->wages*：

```
(test-hours->wages
  (cons 5.65 (cons 8.75 empty))
  (cons 40 (cons 30 empty))
  (cons 226.0 (cons 262.5 empty)))
```

如果在测试中出现了错误，这个包含四个元素的表会被显示，并准确地说明是哪个测试例子出错了。

用 equal?进行测试：*Scheme* 的开发者预计到人们会需要一个一般化的相等性程序，所以提供：

```
;; equal? : any-value any-value -> boolean
;; 判断两个值是否结构相等，
;; 而且在相同的位置上包含相同的原子值
```

当 equal?被作用于两个表时，它使用类似于 *list=?*的方法对它们进行比较；当 equal?被作用于一对结构体时，如果它们是同种类型的结构体，它会比较它们相应的字段；当 equal?被作用于一对原子值时，它使用=、symbol=?、boolean=?，或者一切合适的东西来比较它们。

> **测试原则**
>
> 当需要对值进行比较时，使用 equal?进行测试。

无序表：在某些情况下，我们使用表，但是不考虑其中元素的顺序。对于这种情况，如果我们要判断某个函数调用返回的结果是否包含了正确的元素，很重要的一点就是，要有像 *contains-same-numbers* 这样的函数（参见习题 17.8.4）。

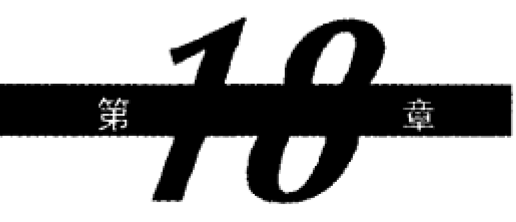
习题

习题 17.8.10 定义一个测试函数，用 equal?测试第 17.1 节中的 *replace-eol-with*，再使用这个函数将例子表达成测试情况。

习题 17.8.11 定义函数 *test-list-pick*，该函数处理第 17.3 节中 *list-pick* 函数的测试。

习题 17.8.12　定义 *test-evaluate*，该函数使用 equal?测试习题 17.7.4 中的 evaluate-*with-defs*。用这个函数重新给出测试。





第18章 局部定义和辖域

程序不仅仅由单一定义组成，许多情况下，程序都会定义许多辅助函数，或者许多相互引用的函数。事实上，随着我们的经验越来越丰富，写出的程序所包含的辅助函数也会越来越多。如果考虑得不够全面，这一大堆函数的集合就会令我们迷失方向。随着函数的体积变大，需要对它们进行组织，使得我们（以及其他读者）可以很快地辨认出程序各个部分之间的关系。

本章介绍 local，这是一个简单的结构，用来组织函数的集合。使用 local，程序设计者可以把互相联系的函数定义聚集到一起，这样读者就能够立即辨认出函数之间的联系。最后，为了介绍 local，我们还必须讨论变量绑定的概念。尽管 Beginning Student Scheme 已经在程序中引入了绑定，但是只有在彻底熟悉这个概念之后，我们才能真正理解 local 定义。

18.1 用 local 组织程序

local 表达式把类似 Definitions 窗口中的任意长的定义序列聚集在一起的。按照惯例，我们先介绍 local 表达式的语法，再介绍其语义，最后介绍它的语用。

local 的语法

local 表达式是另一种类型的表达式：

```
<exp>=(local (<def-1> ...<def-n>) <exp>)
```

与以往一样，*<def-1>*……*<def-n>*是任意长的定义序列（参见图 18.1），而*<exp>*是任意一个表达式。换句话说，local 表达式依次由关键字 local、用“(”和“)”聚合的定义序列以及一个表达式组成。

```
<def> = (define (<var> <var> ...<var>) <exp>)
      | (define <var> <exp>)
      | (define-struct <var> (<var> ...<var>))
```

图 18.1 Scheme 定义

关键字 local 把这种新的表达式类型和其他表达式区分开来，就像 cond 把条件表达式和其他调用区分开一样，local 后面用括号括住的序列叫做局部定义。这种定义被称为局部定义的变量、函数或者结构体。Definitions 窗口中的所有定义被称为最外层定义。对于变量定义或函数定义来说，一个名字最多只能在左部出现一次。定义中的表达式被称为右部表达式。跟在定义后的表达式则被称为主体。

我们来观察一个例子：

```
(local ((define (f x) (+ x 5))
        (define (g alon)
          (cond
          [(empty? alon) empty]
          [else (cons (f (first alon)) (g (rest alon)))])))
```

```
    (g (list 1 2 3)))
```

这里，局部定义的函数是 *f* 和 *g*。前一个函数定义的右部是(+ *x* 5)；后一个函数的右部是：

```
(cond
  [(empty? alon) empty]
  [else (cons (f (first alon)) (g (rest alon)))])
```

最后，local 表达式的主体是(*g* (**list** 1 2 3))。

习题

习题 18.1.1　用红笔划出下列局部定义的变量和函数，用绿笔划出它们的右部，用篮笔划出 local 表达式的主体：

1.
```
(local ((define x (* y 3)))
  (* x x))
```
2.
```
(local ((define (odd an)
          (cond
        [(zero? an) false]
        [else (even (sub1 an))]))
        (define (even an)
        (cond
        [(zero? an) true]
        [else (odd (sub1 an))])))
  (even a-nat-num))
```
3.
```
(local ((define (f x) (g x (+ x 1)))
        (define (g x y) (f (+ x y))))
  (+ (f 10) (g 10 20)))
```

习题 18.1.2　解释下列短语为何不符合语法：

1.
```
(local ((define x 10)
        (y (+ x x)))
  y)
```
2.
```
(local ((define (f x) (+ (* x x) (* 3 x) 15))
        (define x 100)
        (define f@100 (f x)))
  f@100 x)
```
3.
```
(local ((define (f x) (+ (* x x) (* 3 x) 14))
        (define x 100)
        (define f (f x)))
  f)
```

习题 18.1.3　判断下列定义哪些是合法的，哪些是不合法的：

1.
```
(define A-CONSTANT
  (not
  (local ((define (odd an)
          (cond
            [(= an 0) false]
            [else (even (- an 1))]))
```

```
            (define (even an)
              (cond
                [(= an 0) true]
                [else (odd (- an 1))])))
        (even a-nat-num))))
```

2.
```
(+ (local ((define (f x) (+ (* x x) (* 3 x) 15))
           (define x 100)
           (define f@100 (f x)))
       f@100)
   1000)
```

3.
```
(local ((define CONST 100)
        (define f x (+ x CONST)))
      (define (g x y z) (f (+ x (* y z)))))
```

解释为什么这些定义是合法的或者是不合法的。

local 的语义

local 表达式的用途是，为主体表达式的计算定义变量、函数或者结构体。在 local 表达式之外，这些定义没有效果。考虑如下的表达式：

```
(local ((define (f x) exp-1)) exp)
```

它在计算 exp 期间，定义了函数 *f*，exp 的返回值就是整个 local 表达式的返回值。类似地，

```
(local ((define PI 3)) exp)
```

在计算 exp 期间，临时让变量 *PI* 代表 3。

我们可以用一条规则来描述 local 表达式的计算，不过这条规则极其复杂。具体来说，这条规则需要两个手工计算步骤。第一步，必须系统地替换所有局部定义的变量、函数和结构体，使得它们的名字不和那些在 Definitions 窗口中使用过的名字重复；第二步，把整个定义序列移到最外层，其后的处理过程同建立了一个新函数一样。

用符号来表示这个计算规则，就是：

```
def-1 ... def-n
E[(local ((define (f-1 x) exp-1) ... (define (f-n x) exp-n)) exp)]
=
def-1 ... def-n (define (f-1' x) exp-1') ... (define (f-n' x) exp-n')
E[exp']
```

为了简单起见，在这条规则中，local 表达式只定义了单参数函数，不过，很容易把它推广到一般的情况。跟往常一样，序列 *def-1 ... def-n* 代表了最外层定义。

这条规则中的不寻常之处是符号 *E*[exp]，它代表了表达式 exp 和它的上下文 *E*。更明确地说，exp 是下一个必须被计算的表达式；*E* 是它的计算环境。

例如，表达式

```
(+ (local ((define (f x) 10)) (f 13)) 5)
```

是一个加法。在可以计算其结果之前，必须把两个子表达式的数值计算出来。既然第一个子表达式不是数，我们先对它进行计算：

```
(local ((define (f x) 10)) (f 13))
```

此时

```
exp = (local ((define (f x) 10)) (f 13))
E = (+ ... 5)
```

在 local 规则的右部，我们可以看到一些带“'”的名字和表达式。这些名字 f-1'，…，*f-n'*是新的函数名，它们与外层定义中的其他名字都不相同；在表达式 *exp-1'*，…，*exp-n'*中的“'”表明这些表达式与 *exp-1*，…，*exp-n* 相同，但是其中包含了 *f-1'*而不是 *f-1*，以此类推。

local 表达式的计算规则是我们到目前为止所遇到的规则中最复杂的，而且事实上，它也是我们将会遇到的所有规则中最复杂的。规则中的两个步骤都很重要，并且起着不同的作用。要说明它们的用途，最好的方法是使用一系列简单的例子。

规则的第一个部分的作用是，排除在最外层环境中定义的名字和将要插入到外层环境中的名字之间的冲突。考虑如下的例子：

```
(define y 10)
(+ y
   (local ((define y 10)
           (define z (+ y y)))
           z))
```

这个表达式引入一个 y 的局部定义，把 y 与自身相加，从而得到 z，最后返回 z 的值。

local 的非正式描述表明，这个例子的计算结果应该是 30。我们用规则来验证这一点。如果我们简单地把 local 中的定义添加到最外层，那么两个 y 的定义就冲突了。重命名步骤防止这种冲突出现，并且阐明了哪个 y 是哪个：

```
= (define y 10)
  (+ y (local ((define y1 10) (define z1 (+ y1 y1))) z1))
= (define y 10)
  (define y1 10)
  (define z1 (+ y1 y1))
  (+ y z1)
= (define y 10)
  (define y1 10)
  (define z1 20)
  (+ 10 z1)
= (define y 10)
  (define y1 10)
  (define z1 20)
  (+ 10 z1)
= (define y 10)
  (define y1 10)
  (define z1 20)
  (+ 10 20)
```

正如我们所期望的，返回值是 30。

因为 local 表达式可以出现在函数主体的内部，所以重命名非常重要，原因是这个函数可能会被多次调用。下面的第二个例子会说明这一点：

```
(define (D x y)
  (local ((define x2 (* x x))
      (define y2 (* y y)))
    (sqrt (+ x2 y2))))
(+ (D 0 1) (D 3 4))
```

函数 *D* 计算它的两个参数的平方和的平方根。因此，(+ (*D* 0 1) (*D* 3 4))的返回值应该是 6。

在*D*计算其结果的过程中，它引入了两个局部变量：*x2*和*y2*。因为*D*被调用了两次，所以它的主体的修改版本会被计算两次，因此它的局部定义必须被加入到最外层两次。重命名步骤确保无论我们多少次把这些定义提升到最外层，它们都不会相互干扰。这个例子是这样运行的：

```
= (define (D x y)
    (local ((define x2 (* x x))
            (define y2 (* y y)))
      (sqrt (+ x2 y2))))
  (+ (local ((define x2 (* 0 0))
             (define y2 (* 1 1)))
       (sqrt (+ x2 y2)))
     (D 3 4))
```

按照正式的规则，要计算表达式(**D 0 1**)，我们重命名 local 定义，并把它提升（到最外层）：

```
=(define(D x y)
    (local ((define x2 (* x x))
            (define y2 (* y y)))
            (sqrt (+ x2 y2))))
  (define x21 (* 0 0))
  (define y21 (* 1 1))
  (+ (sqrt (+ x21 y21))
     (D 3 4))
```

从这里开始，一直到遇到第二个嵌套的 local 表达式为止，计算都按照标准的规则进行：

```
=(define (D x y)
    (local ((define x2 (* x x))
            (define y2 (* y y)))
     (sqrt (+ x2 y2))))
  (define x21 0)
  (define y21 1)
  (+ 1 (local ((define x2 (* 3 3))
               (define y2 (* 4 4)))
         (sqrt (+ x2 y2))))
= (define (D x y)
    (local ((define x2 (* x x))
            (define y2 (* y y)))
      (sqrt (+ x2 y2))))
  (define x21 0)
  (define y21 1)
  (define x22 9)
  (define y22 16)
  (+ 1 (sqrt (+ x22 y22)))
```

通过再一次重命名*x2*和*y2*，我们避免了冲突。接下去，表达式的计算就很简单了：

```
 (+ 1 (sqrt (+ x22 y22)))
= (+ 1 (sqrt (+ 9 y22)))
= (+ 1 (sqrt (+ 9 16)))
= (+ 1 (sqrt 25))
= (+ 1 5)
```

```
= 6
```

正如我们所期望的，返回值是 6。[1]

习题

习题 18.1.4　既然在计算过程中，local 定义会被添加到 Definitions 窗口，那么，我们也可以期望，通过在 Interactions 窗口中输入变量而看到它们的值。这可能吗？为什么？

习题 18.1.5　手工计算下列表达式：

```
1. (local ((define (x y) (* 3 y)))
     (* (x 2) 5))
2. (local ((define (f c) (+ (* 9/5 c) 32)))
     (- (f 0) (f 10)))
3. (local ((define (odd? n)
             (cond
               [(zero? n) false]
               [else (even? (sub1 n))]))
           (define (even? n)
             (cond
               [(zero? n) true]
               [else (odd? (sub1 n))])))
     (even? 1))
4. (+ (local ((define (f x) (g (+ x 1) 22))
              (define (g x y) (+ x y)))
        (f 10))
      555)
5. (define (h n)
     (cond
       [(= n 0) empty]
       [else (local ((define r (* n n)))
               (cons r (h (- n 1))))]))
   (h 2)
```

计算的过程应该包括所有的 local 化简步骤。

local 的语用(一)

local 表达式最大的用处是封装函数的集合。考虑一个例子，即我们在第 12.2 节中定义的排序函数：

```
;; sort : list-of-numbers -> list-of-numbers
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon) (sort (rest alon)))]))

;; insert : number list-of-numbers (sorted) -> list-of-numbers
(define (insert an alon)
  (cond
```

[1] 随着我们不断地使用这种方法计算表达式，定义的表会变得越来越长。万幸的是，DrScheme 知道怎样来管理这种不断变长的表。实际上，DrScheme 有时候会丢弃不再需要的定义。

```
      [(empty? alon) (list an)]
      [else (cond
        [(> an (first alon)) (cons an alon)]
        [else (cons (first alon) (insert an (rest alon)))])]))
```

第一个定义本身就定义了 sort，而第二个定义定义了一个辅助函数，该辅助函数把一个数插入到数表中。第一个定义使用第二个定义和自然递归来构建返回值，即把表的第一个元素插入到表的其余部分的排序结果中。

这两个定义合起来组成了对数表排序的程序。为了明确指出这两个函数之间的紧密联系，我们可以，也应该使用 local 表达式。具体来说，我们定义程序 *sort*，它直接以辅助函数的形式引入这两个函数：

```
;; sort : list-of-numbers -> list-of-numbers
(define (sort alon)
  (local ((define (sort alon)
            (cond
            [(empty? alon) empty]
            [(cons? alon) (insert (first alon)
           (sort (rest alon)))]))
        (define (insert an alon)
          (cond
            [(empty? alon) (list an)]
            [else (cond
                [(> an (first alon)) (cons an alon)]
                [else (cons (first alon)
                    (insert an (rest alon)))])])))
      (sort alon)))
```

这里，local 表达式的主体简单地把参数传递给局部定义的函数 *sort*。

> **使用 local 的原则**
>
> 遵照设计诀窍，开发一个函数。如果这个函数需要辅助函数，那么用一个 local 表达式把它们聚集起来，再把这个 local 表达式放到一个新的定义中。local 的主体应该把主函数作用到新定义的函数的参数上。

习题

习题 18.1.6 手工计算(*sort* (list 2 1 3))，直到使用到局部定义的 *sort* 函数为止。再对 (equal? (*sort* (list 1)) (*sort* (list 2))) 进行同样的处理。

习题 18.1.7 使用 local 表达式来组织第 10.3 节中移动图片的函数。

习题 18.1.8 使用 local 表达式来组织图 12.2 中绘制多边形的函数。

习题 18.1.9 使用 local 表达式来组织第 12.4 节中重新排列单词的函数。

习题 18.1.10 使用 local 表达式来组织第 15.1 节中寻找蓝眼睛后代的函数。

local 的语用(二)

假设我们需要一个函数，返回某个表中的某个元素的最后一次出现。为了精确起见，假设我们有一些摇滚乐明星记录的表，其中每位明星用两个值表示：

```
(define-struct star (name instrument))
```

> *star*（明星）（记录）是结构体：
>
> (make-star *s t*)
>
> 其中 *s* 和 *t* 是符号。

下面是一个例子：

```
(define alos
  (list (make-star 'Chris 'saxophone)
        (make-star 'Robby 'trumpet)
        (make-star 'Matt 'violin)
        (make-star 'Wen 'guitar)
        (make-star 'Matt 'radio)))
```

在这个表中，'Matt 出现了两次。所以，如果要判断最后一次'Matt 的出现所伴随的是什么乐器，就应该得出'radio。另一方面，对于'Wen 来说，函数应返回'guitar。当然，（在这个表中）寻找'Kate 所对应的乐器会返回 false，这表明没有'Kate 的记录。

先写出合约、用途说明以及头部：

```
;; last-occurrence : symbol list-of-star -> star or false
;; 求出 alostars 中最后一条名字字段包含 s 的记录
(define (last-occurrence s alostars) ...)
```

这个合约很不同寻常，因为它在箭头的右边提到了两种数据类型：*star* 以及 false。虽然我们以前没有看到过这种类型的合约，但是它的含义很明显。这个函数可能返回 *star*，也可能返回 false。

我们已经开发了一些例子，所以可以直接进入设计诀窍中模板开发的阶段：

```
(define (last-occurrence s alostars)
  (cond
    [(empty? alostars) ...]
    [else ... (first alostars) ... (last-occurrence s (rest alostars)) ...]))
```

当然，在我们填写这个模板的空缺时，这个函数真正的问题才会出现。根据说明，第一个子句的答案是 false。我们还不明确怎样形成第二个子句的答案。目前，我们有：

1. (first *alostars*)是给定的表中的第一个 *star* 记录。如果它的名字字段等于 *s*，它可能是最终的结果，也可能不是。这完全取决于表的其余部分中的记录。

2. (*last-occurrence s* (rest *alostars*))计算结果是：*star* 记录，其名字字段为 *s*，或者是 false。在第一种情况下，这个 *star* 记录就是计算的结果；在第二种情况下，计算结果要么是 false，要么是第一个 *record*。

第二点意味着我们需要两次用到自然递归，第一次检查它是 *star* 还是 *boolean*，第二次，如果它是 *star*，就用它作为答案。

最好用 local 表达式来表达自然递归的两次使用：

```
(define (last-occurrence s alostars)
  (cond
    [(empty? alostars) false]
    [else (local ((define r (last-occurrence s (rest alostars))))
            (cond
              [(star? r) r]
            ...))]))
```

嵌套的 local 表达式给自然递归的结果起一个名字，cond 表达式则两次使用到这个名字。通过用右部来代替 *r*，我们可以消去这个 local 表达式：

```
(define (last-occurrence s alostars)
  (cond
    [(empty? alostars) false]
    [else (cond
            [(star? (last-occurrence s (rest alostars)))
           (last-occurrence s (rest alostars))]
            ...)]))
```

但是，有两个自然递归的函数可读性很差，使用 local 的函数版本更为优越。

只需一个小步骤，就可以从模板得到完整定义：

```
;; last-occurrence : symbol (list-of-star) -> star or false
;; 求出 alostars 中最后一条名字字段包含 s 的记录
(define (last-occurrence s alostars)
  (cond
    [(empty? alostars) false]
    [else (local ((define r (last-occurrence s (rest alostars))))
            (cond
              [(star? r) r]
              [(symbol=? (star-name (first alostars)) s) (first alostars)]
              [else false]))]))
```

在嵌套的 cond 表达式中，如果 *r* 不是 *star* 记录的话，第二个子句对第一条记录的 *name* 字段和 *s* 进行比较。在这情况下，表的其余部分中没有记录的名字能够匹配，那么，如果第一条记录的名字能够匹配，它就是计算结果；否则，整个表就不包含我们正在寻找的名字，所以计算结果就是 false。

习题

习题 18.1.11 手工计算下面的测试：

```
(last-occurrence 'Matt
  (list (make-star 'Matt 'violin)
        (make-star 'Matt 'radio)))
```

有多少个 local 表达式被提升（到最外层）？

习题 18.1.12 考虑下面的函数定义：

```
;; max : non-empty-lon -> number
;; 求出 alon 中最大的数
(define (max alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (cond
         [(> (first alon) (max (rest alon))) (first alon)]
         [else (max (rest alon))])]))
```

在嵌套的 cond 表达式中，两个子句都计算了(*max* (rest *an-inv*))，所以我们自然可以用 local 表达式来处理它。用下面的表来测试两个版本的 *max*，并解释结果。

(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)

习题 18.1.13　开发函数 *to-blue-eyed-ancestor*。这个函数读入一棵家谱树（*ftn*）（参见第 14.1 节），返回一个表，该表说明如何到达一个蓝眼睛的祖先。如果树中没有蓝眼睛的祖先，函数返回 false。

函数的合约、用途说明以及头部如下：

```
;; to-blue-eyed-ancestor : ftn -> path or false
;; 计算从 a-ftn 树到一个蓝眼睛祖先的路径
(define (to-blue-eyed-ancestor a-ftn) ...)
```

路径是'father 和'mother 的表，我们把'father 和'mother 称为方向。它们的数据定义是：

direction（方向）是下列两者之一：
1. 符号'father。
2. 符号'mother。

path（路径）是下列两者之一：
1. empty。
2. (cons *s los*)，其中 *s* 是方向，*los* 是路径。

空路径代表 *a-ftn* 的 *eyes* 字段就是'blue。如果路径的第一个元素是'mother，我们就可以在母亲的家谱树中使用其余路径搜索蓝眼睛祖先。类似地，如果路径的第一个元素是'father，我们就在父亲的家谱树中使用其余路径作进一步的搜索。

例子：

1. 对于图 14.1 中的家谱树，(*to-blue-eyed-ancestor Gustav*)返回(list 'mother)；
2. 对于同一棵家谱树，(*to-blue-eyed-ancestor Adam*)返回 false；
3. 如果我们加上定义(define *Hal* (make-child *Gustav Eva* 'Gustav 1988 'hazel))，那么(*to-blue-eyed-ancestor Hal*)会返回(list 'father 'mother)。

用这些例子建立测试。使用第 17.8 节的方法，用布尔值表达式表示测试。

回溯：函数 *last-occurrence* 以及 *to-blue-eyed-ancestor* 返回两种类型的结果：一种结果表明搜索成功，另一种结果表明搜索失败。这两个函数都是递归的。如果某个自然递归不能找出所需的结果，函数会试着用另一种方法计算结果。事实上，*to-blue-eyed-ancestor* 会使用另一个自然递归。

这种计算答案的策略是回溯的一种简单形式。到目前为止，在我们所处理的数据中，回溯还算是简单的，它是一种节省计算步骤的方法。我们经常可以写出两个单独的递归函数，它们都可以完成同样的目的，如同这个回溯函数。

在第 28 章中，我们会进一步研究回溯。另外，我们会在第 29 章中讨论计算步骤的计算。

习题 18.1.14　用回溯的观点讨论习题 15.3.4 中的函数 *find*。

local 的语用(三)

考虑如下的函数定义：

```
;; mult10 : list-of-digits -> list-of-numbers
;; 用(expt 10 p)去乘 alod 中的每一个数位，
;; 其中 p 是该数位之后数位的数量，从而建立一个数表
(define (mult10 alod)
  (cond
    [(empty? alod) 0]
    [else (cons (* (expt 10 (length (rest alod))) (first alod))
                (mult10 (rest alod)))]))
```

下面是一个测试：

```
(equal? (mult10 (list 1 2 3)) (list 100 20 3))
```

显然，这个函数可以用来把数字表转化为一个数。

mult10 的定义有一小问题，在第二个子句中，它需要计算结果中的第一个元素。这是一个很长的表达式，而且并不完全与用途说明相一致。通过在第二个子句中使用 local 表达式，我们可以为计算过程中的中间结果引入一个名字：

```
;; mult10 : list-of-digits -> list-of-numbers
;; 用(expt 10 p)去乘 alod 中的每一个数位，
;; 其中 p 是该数位之后数位的数量，从而建立一个数表
(define (mult10 alon)
  (cond
    [(empty? alon) empty]
    [else (local ((define a-digit (first alon))
                  (define p (length (rest alon))))
            ;;________________________________________________
            (cons (* (expt 10 p) a-digit) (mult10 (rest alon))))]))
```

在阅读这个定义时，这些名字可以帮助我们理解表达式。

当某个值需要多次计算时，使用 local 是最合适的。例如，*mult10* 中的表达式(rest *alon*)。通过为重复的表达式引入名字，我们还可以避免 DrScheme 的某些（少量的）耗费：

```
(define (mult10 alon)
  (cond
    [(empty? alon) empty]
    [else (local ((define a-digit (first alon))
                  (define the-rest (rest alon))
                  (define p (length the-rest)))
            ;; ------------------------------------------------------
            (cons (* (expt 10 p) a-digit) (mult10 the-rest)))]))
```

对于我们已经开发的程序来说，local 的这第三种用途几乎没有什么用处。使用辅助函数总能得到更好的效果。不过，在本书的后续部分，我们会遇到许多不同类型的函数，它们可以使用 local 表达式。在某些情况下，必须使用 local 表达式，就如同 *mult10*。

习题

习题 18.1.15 考虑函数定义：

```
;; extract1 : inventory -> inventory
;; 用 an-inv 中所有价格低于 1 元的元素建立存货清单
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) empty]
    [else (cond
            [(<= (ir-price (first an-inv)) 1.00)
             (cons (first an-inv) (extract1 (rest an-inv)))]
            [else (extract1 (rest an-inv))])]))
```

在这个嵌套的 cond 表达式中，两个子句都从 *an-inv* 中提取了第一个元素，也都计算了(*extract1* (rest

an-inv))。

为这些表达式引入 local 表达式。

18.2 辖域和块结构

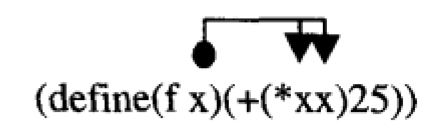
要引入 local，还需要一些关于 Scheme 语法和函数结构的术语。具体来说，我们需要有词汇来讨论变量、函数和结构体的名字的使用。作为一个简单的例子，考虑如下两个定义：

```
(define (f x) (+ (* x x) 25))
(define (g x) (* 12 (expt x 5)))
```

显然，*f* 中带下划线的 *x* 出现与 *g* 中的 *x* 完全没有关系。正如以前所提到的，如果系统地把所有带下划线的 *x* 都替换成 *y*，这个函数还是可以计算出相同的值。简而言之，带下划线的 *x* 出现仅仅在 *f* 的定义中有意义，而在其他任何地方都没有意义。

尽管如此，*x* 的第一个出现还是与其他出现不同。当把 *f* 作用于数 *n*，这个 *x* 就完全消失了；相反，另两个 *x* 被替换成了 *n*。为了区分这两种形式的变量出现，我们把函数名右边变量 *x* 的出现称为绑定出现，而把在函数主体中变量 *x* 的出现称为被绑定出现。我们还说，*x* 的绑定出现绑定了 *f* 的主体中所有 *x* 的被绑定出现。在上述的讨论中，显然 *f* 的主体是函数唯一的区域，只有其中 *x* 带下划线的绑定出现才能绑定其他的 *x*。这个区域被称为 *x* 的辖域。我们还说，*f* 和 *g*（或者其他位于 Definitions 窗口中的定义）拥有全局辖域，有时候也称为自由出现。

把 *f* 作用于数 *n* 的描述表明，定义的图形表示法是：



在第一个 *x* 上方的圆点表明它是绑定出现。从该圆点出发的箭头表示了值的流动。也就是说，知道了绑定出现的值，也就知道了被绑定出现的值。换一种说法，在计算中，我们知道了某个变量的绑定出现在哪里，也就知道了它的值会从哪里来。

沿着相同的连线，变量的辖域还指明了我们可以在哪里对它进行重命名。如果我们要重命名某个参数，比方说，把 *x* 重命名为 *y*，就要搜索该参数辖域内所有的被绑定出现，把它们替换成 *y*。例如，如果函数定义为：

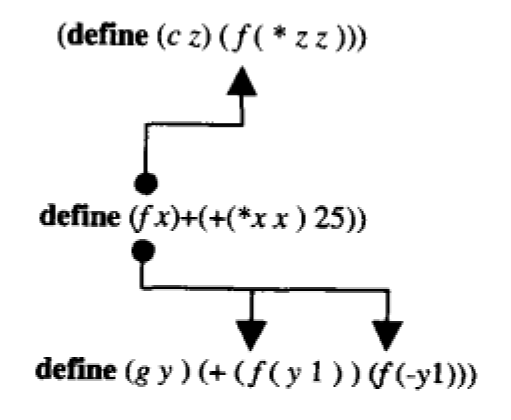
```
(define (f x) (+ (* x x) 25))
```

那么把 x 重命名为 y 就影响了两个被绑定出现：

```
(define (f y) (+ (* y y) 25))
```

在定义之外，没有其他 *x* 的出现需要被修改。

显然，函数定义还引入了函数名的绑定出现。如果某个定义引入了一个名为 *f* 的函数，*f* 的辖域就是整个定义序列：



也就是说，*f*的辖域包括了在*f*之前和之后的所有定义。

习题

习题 18.2.1 下面是一段简单的 Scheme 程序：

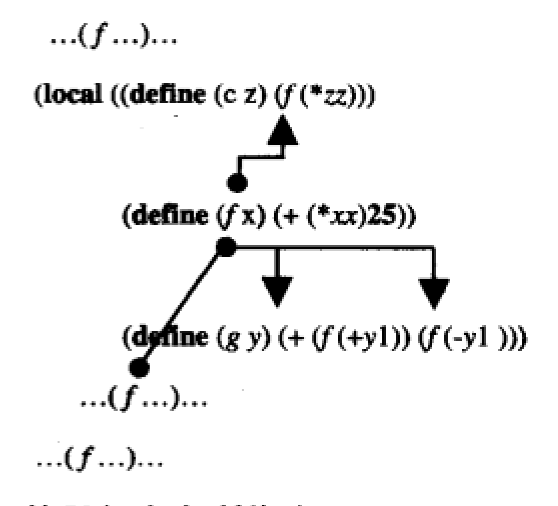
```
(define (p1 x y)
  (+ (* x y)
     (+ (* 2 x)
        (+ (* 2 y) 22))))
(define (p2 x)
  (+ (* 55 x) (+ x 11)))
(define (p3 x)
  (+ (p1 x 0)
     (+ (p1 x 1) (p2 x))))
```

画出所有从 *p1* 的参数 *x* 到其被绑定出现的箭头。画出从 *p1* 到达其所有被绑定出现的箭头。

复制这个函数，并把 *p1* 的参数 *x* 重命名为 *a*，把 *p3* 的参数 *x* 重命名为 *b*。

用 DrScheme 的 Check Syntax 按钮检查这个结果。

与最外层函数定义不同，在 local 中，函数定义的辖域是有限的。具体来说，局部定义的辖域就是该 local 表达式。考虑在 local 表达式中某个辅助函数*f*的定义，它绑定了 local 表达式中的所有*f*，而不涉及其外的*f*：



在 local 之外的两个*f*没有被*f*的局部定义所绑定。

在任何时候，对于函数定义来说，不论是不是局部函数，其参数都只在函数主体中绑定，不影响其他地方：

(local ((define (*f x*) (+ (* *x x*) 25))

...*x*...)

既然函数名称或者函数参数的辖域是一个区域，人们通常用一个方框来表示辖域。更精确地说，对于参数，我们在函数主体上画一个方框：

(define(*f*x)

(+(*2x)10)

对于局部定义，我们围绕整个 local 表达式画一个方框：

```
(define(h z)
 (local((define(f x)(+(*x x)55))
        (define(g y)(+f y)10)))
 (f z))
)
```

在这个例子中，方框表示 *f* 和 *g* 定义的辖域。

使用方框或者辖域，我们还可以方便地理解，在 local 表达式内重用某个函数名意味着什么：

```
(define(a-function y)
 local((define(f x y)(+*x y)(+x y)))
 (
    (define(g z)
     local((define(f x)(+(x x)55))
      (
       define(g y)(+f y)10)))
      (f z)
    )
    (define(h x)(f x(g x)))
  (h y))
)
```

内层的方框表示里面的 *f* 定义的辖域；外层的方框是外面的 *f* 定义的辖域。相应地，所有在内层方框中 *f* 的出现都引用内层的 local；所有在外层方框中 *f* 的出现都引用外层的 local。换句话说，外面的 *f* 的定义的辖域有一个缺口：内层的方框，也就是里层 *f* 定义的辖域。

参数定义的辖域中也可以出现缺口。下面是一个例子：

(define(*f x*)



在这个函数中，两次用到了参数 *x*：函数 *f* 和函数 *g* 都使用它作参数。*g* 的辖域被嵌套在 *f* 的辖域之内，因此，外层 *x* 的使用辖域就有一个空缺。

一般来说，如果在一个函数中某个名字多次出现，那么描述相应辖域的方框决不会重叠。在某些情况下，某个方框会嵌套在其他方框中，从而形成空缺。不过，方框的图片总是有层次的，嵌套的方框总是一个比一个小。

习题

习题 18.2.2　下面是一个简单的 Scheme 函数：

```
;; sort : list-of-numbers -> list-of-numbers
(define (sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) empty]
```

```
                [(cons? alon) (insert (first alon) (sort (rest alon)))]))
        (define (insert an alon)
        (cond
         [(empty? alon) (list an)]
         [else (cond
              [(> an (first alon)) (cons an alon)]
              [else (cons (first alon) (insert an (rest alon)))])]))))
    (sort alon)))
```

画出围绕每一个绑定变量 *sort* 和 *alon* 的辖域的方框。然后画出从每一个 *sort* 出发、指向相应的被绑定出现的箭头。

习题 18.2.3 回忆一下，每个变量的出现都从它相应的绑定变量获取值。考虑如下的定义：

```
(define x (cons 1 x))
```

带下划线的 *x* 出现被绑定到哪里？因为这是变量定义，而不是函数定义，所以，如果要使用这个函数，我们就需要计算其右部。按照我们的规则，右部的值是什么？

第四部分

抽象设计







定义的相似性

许多数据或函数的定义都很相似，例如，符号表的定义和数表的定义只有两处不同：一处是数据类型名，另一处是关键字“symbol”和“number”。同样，我们几乎无法区分从符号表中寻找某个特定符号的函数与从数表中寻找某个特定数的函数。

许多程序错误出现的原因都是重复，所以好的程序设计者总是尽可能避免重复。在开发一组函数的时候，特别是开发源于同一个模板的一组函数时，我们能很快地辨认出它们之间的类似处。然后就可以修改这些函数，尽可能地去除重复。换句话说，函数就好比是散文、备忘录、小说或者是其他作品，第一稿只是草稿，除非数易其稿，否则不可能简单明了地表达作者的思想并娱乐读者。因为我们所写的函数将会被许多人阅读，并且可能有人在读懂后对它们进行改进，所以我们必须学会“编辑”函数，使其更完善。

在编辑程序的过程中，最主要的步骤是去除重复。在这一章中，我们讨论函数的类似之处以及数据定义的类似之处，并且考虑如何消除它们。这里讨论的消除类似的方法仅针对如 Scheme 这样的函数式程序设计语言；不过，其他类型的语言，特别是面向对象语言，也支持类似的机制，其一般被为模式。

19.1 函数的类似之处

遵照设计诀窍，通过数据定义和输入，我们可以完全确定函数的模板，也就是函数的基本结构。实际上，模板表达了对输入数据的了解。显然，读入相同数据类型的函数看起来很类似。

```
;; contains-doll? : los -> Boolean
;; 测定 alos 是否含有符号'doll
(define (contains-doll? alos)
  (cond
    [(empty? alos) false]
    [else
      (cond
[(symbol=? (first alos) 'doll) true]
[else
(contains-doll? (rest alos))])]))
```

```
;; contains-car? : los -> boolean
;; 测定 alos 是否含有符号'car
(define (contains-car? alos)
  (cond
    [(empty? alos) false]
    [else
      (cond
    [(symbol=? (first alos) 'car) true]
    [else
      (contains-car? (rest alos))])]))
```

图 19.1　两个类似的函数

请仔细观察图 19.1 中的两个函数，它们的功能是在符号表（含有玩具的名字）中寻找特定的玩具，左边的函数寻找'doll，右边的函数寻找'car。这两个函数几乎无法相互区别：每个函数都是由一个含有两个子句的 cond 表达式组成：当输入是 empty 时，两个函数都返回 false；两个函数都使用第二个嵌套的 cond 表达式来判断表的第一个元素是否就是要找的东西。两个函数唯一的不同是，在第二个嵌套的 cond 表达式中，它们所比较的东西不同，*contains-doll?*使用'doll 而 *contains-car?*使用'car。为了突出这个不同

点，在图中这两个符号用方框框住了。

好的程序设计者不会定义多个类似的函数，而是定义单一的、在一个表中既能够寻找'doll，又能够寻找'car 的函数。这个更一般的函数需要额外的参数，即我们需要寻找的符号：

```
;; contains? : symbol los -> Boolean
;; 测定 alos 是否含有符号 s
(define (contains? s alos)
  (cond
    [(empty? alos) false]
    [else (cond
            [(symbol=? (first alos) s)
              true]
              [else
                (contains? s (rest alos))])]))
```

现在，只要把 *contains?*函数作用于'doll 和一个符号表上，就可以在表中寻找'doll 了。同时，*contains?*函数也可以寻找其他的符号。定义这样一个函数同时解决了许多相关的问题。

将两个相关的函数结合成一个单独函数的过程叫做函数抽象[1]。定义高度抽象的函数有许多好处。第一个好处是，这样一个单独的函数可以同时完成很多任务。在这个例子中，*contains?*函数可以查找许多不同的符号，而不是只能查找某个特定的符号。

```
;; below : lon number -> lon
;; 构造一个表，含有 alon 中
;; 比 t 大的成员
(define (below alon t)
  (cond
    [(empty? alon) empty]
    [else
      (cond
    [(< (first alon) t)
      (cons (first alon)
       (below (rest alon) t))]
    [else
      (below (rest alon) t)])]))
```

```
;; above : lon number -> lon
;; 构造一个表，含有 alon 中
;; 比 t 小的成员
(define (above alon t)
  (cond
    [(empty? alon) empty]
    [else
      (cond
    [(> (first alon) t)
      (cons (first alon)
            (above (rest alon) t))]
      [else
        (above (rest alon) t)])]))
```

图 19.2 另外两个类似的函数

就 *contains-doll?*和 *contains-car?*而言，函数抽象一点也不好玩。但是，我们会遇到更有趣的情形，例如，图 19.2 中的函数。左边的函数的参数是一个数表和一个下界，生成所有比下界大的数组成的表；而右边的函数的参数是一个数表和一个上界，生成含有所有比上界小的数的表。

这两个函数之间的区别是比较运算符，左边的函数使用小于号，而右边的函数使用大于号。和前一个例子类似，我们对这两个函数进行抽象，使用额外的参数来确定具体的关系算子是 *below* 还是 *above*：

```
(define (filter1 rel-op alon t)
  (cond
```

[1] “抽象”这个术语来源于数学。数学家把“6”称作为一个抽象的数，因为它代表了为六样东西命名的所有方法。相反，“6 厘米”和“6 只鸡蛋”才是“6”的具体实例，因为它们分别表达了尺寸和数量。

```
    [(empty? alon) empty]
    [else (cond
        [ (rel-op (first alon) t)
         (cons (first alon)
               (filter1 rel-op (rest alon) t))]
        [else
         (filter1 rel-op (rest alon) t)])]))
```

要想使用这个函数，我们必须指定三个参数：用于比较两个数的关系算子 *R*，以及数表 *L* 和数 *N*。然后，这个函数生成 *L* 中所有(*R i N*)的计算结果是 true 的 *i* 的组成表。现在，我们还不知道如何写出像 *filter1* 这样的函数的合约。我们先跳过有关合约的问题，在 20.2 节中再来讨论这个问题。

我们通过一个例子来观察 *filter1* 是怎样工作的。显然，只要输入的表是 empty，无论其他参数是什么，返回的结果总是 empty。

```
  (filter1 < empty 5)
= empty
```

接着再来观察一个稍微复杂一点的例子：

(*filter1* < (cons 4 empty) 5)

因为表中唯一的元素是 4，同时(< 4 5)为 true，所以计算的结果应当是(cons 4 empty)。

程序运行的第一步基于调用规则：

```
  (filter1 < (cons 4 empty) 5)
= (cond
    [(empty? (cons 4 empty)) empty]
    [else (cond
            [(< (first (cons 4 empty)) 5)
           (cons (first (cons 4 empty))
                 (filter1 < (rest (cons 4 empty)) 5))]
           [else (filter1 < (rest (cons 4 empty)) 5)])])
```

也就是说，系统将 *filter1* 的主体中所有的 *rel-op* 替换成<，*t* 替换成 **5**，*alon* 替换成(cons 4 empty)。其余的计算过程就很直观了：

```
  (cond
    [(empty? (cons 4 empty)) empty]
    [else (cond
            [(< (first (cons 4 empty)) 5)
            (cons (first (cons 4 empty))
                  (filter1 < (rest (cons 4 empty)) 5))]
            [else (filter1 < (rest (cons 4 empty)) 5)])])

= (cond
    [(< (first (cons 4 empty)) 5)
     (cons (first (cons 4 empty))
           (filter1 < (rest (cons 4 empty)) 5))]
    [else (filter1 < (rest (cons 4 empty)) 5)])

= (cond
    [(< 4 5) (cons (first (cons 4 empty))
```

```
                    (filter1 < (rest (cons 4 empty)) 5))]
    [else (filter1 < (rest (cons 4 empty)) 5)])

 = (cond
    [true (cons (first (cons 4 empty))
                (filter1 < (rest (cons 4 empty)) 5))]
    [else (filter1 < (rest (cons 4 empty)) 5)])

 = (cons 4 (filter1 < (rest (cons 4 empty)) 5))
 = (cons 4 (filter1 < empty 5))
 = (cons 4 empty)
```

在上面的计算中，最后一个等式就是我们先前讨论的例子。

最后一个例子是将 *filter1* 作用于包含两个元素的表：

```
  (filter1 < (cons 6 (cons 4 empty)) 5)
 = (filter1 < (cons 4 empty) 5)
 = (cons 4 (filter1 < empty 5))
 = (cons 4 empty)
```

其中，唯一的新步骤是第一步。*filter1* 判断出表中的第一元素并不小于上限，所以它并没有被放入自然递归的结果中。

习题

习题 19.1.1 使用手工计算，一步一步地验证下面的等式：

```
  (filter1 < (cons 6 (cons 4 empty)) 5)
 = (filter1 < (cons 4 empty) 5)
```

习题 19.1.2 手工计算下面的表达式：

```
  (filter1 > (cons 8 (cons 6 (cons 4 empty))) 5)
```

只需给出主要步骤。

计算结果表明，*(filter1 < alon t)*给出的结果与*(below alon t)*完全相同，那正是我们想要得到的结果。按照相同的理由，*(filter1 > alon t)*产生的结果与*(above alon t)*也是一样的。所以假设我们给出如下的定义：

```
;; below1 : lon number -> lon
(define (below1 alon t)
  (filter1 < alon t))
```

```
;; above1 : lon number -> lon
(define (above1 alon t)
  (filter1 > alon t))
```

显然，below1 会产生和 below 相同的结果，而 above1 会产生和 above 相同的结果。简而言之，通过使用 *filter1*，我们只使用一行代码就可以给出 *below* 和 *above* 的定义。

更好的是：有了像 *filter1* 这样的抽象函数，我们还可以将它移作它用。下面就是 *filter1* 另外的三个用途：

1. *(filter1 = alon t)* ：这个表达式提取出 *alon* 中所有的等于 *t* 的数。
2. *(filter1 <= alon t)* ：这个表达式生成含有 *alon* 中所有的小于等于 *t* 的数的表。
3. *(filter1 >= alon t)* ：最后这个表达式给出含有 *alon* 中所有的大于等于限值的数的表。

一般来说，*filter1* 的第一个参数并非一定要是 Scheme 预定义的操作；它可以是任意一个读入两个参数、返回布尔值的函数。考虑如下的例子：

```
;; squared>? : number number -> boolean
(define (squared>? x c)
  (> (* x x) c))
```

当边长为 *x* 的正方形的面积大于限值 *c* 时，这个函数给出结果 true，或者说，这个函数判断 $x^2>c$ 是否成立。现在，我们可以把 *filter1* 作用于这个函数和一个数表上：

```
(filter1 squared>? (list 1 2 3 4 5) 10)
```

该表达式从表(list 1 2 3 4 5)中抽取出所有平方比 10 大的数。

简单的手工计算如下（这里只给出开头部分）：

```
  (filter1 squared>? (list 1 2 3 4 5) 10)
= (cond
    [(empty? (list 1 2 3 4 5)) empty]
    [else (cond
        [(squared>? (first (list 1 2 3 4 5)) 10)
         (cons (first (list 1 2 3 4 5))
           (filter1 squared>? (rest (list 1 2 3 4 5)) 10))]
        [else
          (filter1 squared>? (rest (list 1 2 3 4 5)) 10)])])
```

上面这一步是使用标准的调用规则，接下来：

```
= (cond
    [(squared>? 1 10)
     (cons (first (list 1 2 3 4 5))
       (filter1 squared>? (rest (list 1 2 3 4 5)) 10))]
    [else
     (filter1 squared>? (rest (list 1 2 3 4 5)) 10)])

= (cond
    [false
     (cons (first (list 1 2 3 4 5))
       (filter1 squared>? (rest (list 1 2 3 4 5)) 10))]
    [else
     (filter1 squared>? (rest (list 1 2 3 4 5)) 10)])
```

最后这一步涉及到 *squared>?*函数，这里我们可以跳过：

```
= (filter1 squared>? (list 2 3 4 5) 10)
= (filter1 squared>? (list 3 4 5) 10)
= (filter1 squared>? (list 4 5) 10)
```

剩余的计算过程留作习题。

习题

习题 19.1.3　使用手工计算证明：

```
  (filter1 squared>? (list 4 5) 10)
= (cons 4 (filter1 squared>? (list 5) 10))
```

只需把 *squared>?*看作为基本操作。

习题 19.1.4　函数 *squared>?*的用途表明，如下函数也能运行：

```
;; squared10? : number number -> boolean
```

```
(define (squared10? x c)
  (> (square x) 10))
```

换句话说，函数 *filter1* 使用的关系函数可以忽略它的第二个参数。毕竟，我们已经知道这个事实，并且，在计算(*filter1 squared>? alon t*)的过程中，它也一直在起作用。

这意味着对 *filter* 函数的另一种简化：

```
(define (filter predicate alon)
  (cond
    [(empty? alon) empty]
    [else (cond
        [(predicate (first alon))
         (cons (first alon)
           (filter predicate (rest alon)))]
        [else
         (filter predicate (rest alon))])]))
```

这个 *filter* 函数只读入关系函数 *predicate* 和一个数表，表中的每一个元素 *i* 都用 *predicate* 函数来检查，如果(*predicate i*)成立，*i* 就被包含在结果中；反之，*i* 就不出现在结果中。

使用 *filter* 函数来定义与 *below* 和 *above* 等价的函数。测试你的定义。

到目前为止，我们已经看到，抽象的函数定义比特定的函数定义更为灵活，并且用途也更广泛。抽象函数定义的第二个、实际上也更为重要的优点是，我们能使用一个函数来处理很多问题。考虑图 19.3 中函数 *filter1* 的两个变体。第一个函数抹平了嵌套的 cond 表达式，这是有经验的程序设计者所希望的。第二个函数使用 local 表达式，从而使嵌套的 cond 表达式更为易读。

```
(define (filter1 rel-op alon t)
  (cond
    [(empty? alon) empty]
    [(rel-op (first alon) t)
     (cons (first alon)
       (filter1 rel-op (rest alon) t))]
    [else
      (filter1 rel-op (rest alon) t)]))
```

```
(define (filter1 rel-op alon t)
  (cond
    [(empty? alon) empty]
    [else
     (local ((define first-item (first alon))
         (define rest-filtered
           (filter1 rel-op (rest alon) t)))
           (cond
       [(rel-op first-item t)
         (cons first-item rest-filtered)]
       [else
      rest-filtered]))]))
```

图 19.3 函数 *filter1* 的两个变体

尽管所有这些改变都很琐碎，但关键的问题是，所有使用到 *filter1* 的地方，包括函数 *below1* 和 *above1* 的定义，都受益于这些改变。类似地，如果我们修正一个逻辑上的错误，所有使用该函数的东西都会被改进。最终，我们甚至可以在抽象函数中加入新的功能，例如，加入一种能够统计出多少元素被滤去的机制。在这种情况下，所有使用这个函数的地方都能受益。在后续的章节中，我们会遇到这种改进。

习题

习题 19.1.5 将如下的两个函数抽象成一个函数：

```
;; min : nelon -> number
```

```
;; max : nelon -> number
```

```
;; 找到 alon 中最小的数
(define (min alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (cond
        [(< (first alon)
         (min (rest alon)))
         (first alon)]
        [else
         (min (rest alon))])]))
```

```
;; 找到 alon 中最大的数
(define (max alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (cond
        [(> (first alon)
         (max (rest alon)))
         (first alon)]
        [else
         (max (rest alon))])]))
```

这两个函数都读入一个非空的数表，左边的函数返回表中最小的数，而右边的函数返回最大的数。使用该抽象函数来定义函数 *min1* 和 *max1*，然后使用下面的三个表对它们进行测试：

```
1. (list 3 7 6 2 9 8)
2. (list 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
3. (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
```

对于比较长的表，为什么这两个函数运行起来非常慢？

改进该抽象函数。首先，引入一个局部名称来表示自然递归的结果。然后，引入一个局部的辅助函数来找出我们所“感兴趣”的东西。使用相同的输入再对 *min1* 和 *max1* 进行测试。

习题 19.1.6　回忆 *sort* 函数的定义。该函数读入一个表，返回排序后的表：

```
;; sort : list-of-numbers -> list-of-numbers
;; 生成降序排列的表，含有 alon 中所有的元素
(define (sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) empty]
              [else (insert (first alon) (sort (rest alon)))]))
           (define (insert an alon)
            (cond
               [(empty? alon) (list an)]
               [else (cond
                        [(> an (first alon)) (cons an alon)]
                        [else (cons (first alon) (insert an (rest alon)))])])))
    (sort alon)))
```

定义 *sort* 的抽象函数，该函数读入比较运算符以及一个数表。使用 *sort* 的抽象函数分别对表(list 2 3 1 5 4)进行升序和降序排列。

19.2　数据定义的类似之处

观察如下的两个数据定义：

list of numbers（数表）是下列两者之一：

1. **empty**。

list of IRs（IR 表）是下列两者之一：

1. **empty**。

2.(**cons** *n l*)
其中 *n* 是数而
l 是数表。

2.(**cons** *n l*)
其中 *n* 是数而
l 是 *IR* 表。

两者都定义了一个表。左边给出了数表的定义；右边描述了存货记录表。其中存货记录是用结构体来表示的，其结构和数据定义如下：

```
(define-struct ir (name price))
```

> *IR* 是结构体：
>
> (**make-ir** *n p*)
>
> 其中 *n* 是符号而 *p* 是数。

既然这两个数据的定义很类似，那么读入它们的函数也是类似的。观察一下图 19.4 中的例子，左边的 *below* 函数过滤表中的数，而右边的 *below-ir* 函数提取出存货记录中价格低于某个限值的记录。这两个函数除了名字必然是不同的以外，只有一处不同：关系算子。

```
;; below : number lon -> lon
;; 构造含有 alon 中所有比 t 小的数的表
t 小的数的表
(define (below alon t)
  (cond
    [(empty? alon) empty]
    [else (cond
            [(< (first alon) t)
             (cons (first alon)
               (below (rest alon) t))]
            [else
             (below (rest alon) t)])]))
```

```
;; below-ir : number loIR -> loIR
;; 构造含有 aloir 中所有价格
;; 比 t 小的记录的表
(define (below aloir t)
  (cond
    [(empty? aloir) empty]
    [else (cond
            [(<ir (first aloir) t)
             (cons (first aloir)
               (below-ir (rest aloir) t))]
            [else
             (below-ir (rest aloir) t)])]))

;; <ir: IR number boolean
(define(<ir ir p)
(<(ir-price ir)))
```

图 19.4　两个类似的函数

如果抽象这两个函数，显然可以得到函数 *filter1*。反过来，我们可以使用 *filter1* 来定义函数 *below-ir*：

```
(define (below-ir1 aloir t)
  (filter1 <ir aloir t))
```

毫不令人惊讶的是，我们可以找到 *filter1* 的另一个用途——毕竟，我们已经提到过抽象函数可以复用到其他用途上。在下面这个例子中，*filter1* 不仅可以过滤数表，它也可以过滤任意的东西——只要我们定义一个函数，能够将这样东西与数进行比较。

实际上，我们所需要的是这样一个函数，它能够将表中的元素与我们传给 *filter1* 的第二个参数进行比较。下面的函数能够从存货记录表中抽取出所有有着相同标签的元素：

```
;; find : loIR symbol -> boolean
;; 判断 aloir 中是否含有记录 t
(define (find aloir t)
  (cons? (filter1 eq-ir? aloir t)))

;; eq-ir? : IR symbol -> boolean
;; 将 ir 的名字与 p 进行比较
```

```
(define (eq-ir? ir p)
  (symbol=? (ir-name ir) p))
```

这个新定义的关系算子能够比较存货记录的名字是否和另一个符号相同。

习题

习题 19.2.1 分别使用手工以及利用 DrScheme 计算下面表达式的值：

1. (*below-ir1* 10 (list (make-ir 'doll 8) (make-ir 'robot 12)))
2. (*find* 'doll (list (make-ir 'doll 8) (make-ir 'robot 12) (make-ir 'doll 13)))

只需给出有关 *filter1* 函数的计算步骤。

简而言之，*filter1* 函数统一考虑了多种输入数据类型。“统一”意味着如果函数 *filter1* 作用于一个 *X* 表——无论 *X* 是什么数据类型，得到的结果总是另一个 *X* 表。这样的函数被称为多态函数，或者叫做一般的函数。

当然，*filter1* 并不是唯一能够处理任意类型的表的函数。许多其他的函数也能够处理不同类型的表。下面的两个函数分别可以测定数表的长度和 *IR* 表的长度：

```
;; length-lon : lon -> number
(define (length-lon alon)
  (cond
    [(empty? alon) empty]
    [else
      (+ (length-lon (rest alon)) 1)]))
```

```
;; length-ir : loIR -> number
(define (length-ir alon)
  (cond
    [(empty? alon) empty]
    [else
     (+ (length-ir (rest alon)) 1)]))
```

这两个函数仅仅是名字不同。如果我们给它们起相同的名字，例如 *length*，那么，它们就完全一样了。

为了写出像 *length* 这样的函数的精确合约，需要使用带参数的数据定义，我们称之为参数数据定义。参数数据定义并不指定数据类型，而是使用变量来表明任意一种 Scheme 数据都能被用在该处。粗略地来说，参数数据定义把引用抽象成一个特定的数据类型集，就如同把一个特定的值抽象成函数。

下面就是 *ITEM* 表的参数定义：

> *List of ITEM*（ITEM 表）是下列二者之一：
> 1. empty。
> 2. (cons *s l*) 其中 *s* 是 ITEM，*l* 是 ITEM 表。

记号 *ITEM*（元素）是一种类型变量，代表任意 Scheme 数据的集合，包括：符号、数、布尔值、*IR*……。通过把 *ITEM* 替换成某个数据的名称，我们就能得到用该抽象数据定义方法定义的表的一个具体实例，这个表可以包含符号、数、布尔值、IR 等元素。为了使合约的语言更加精确，我们引入一个新的缩写：

(listof *ITEM*)

我们用(listof *ITEM*)这个名字来表示如上所示的抽象数据定义。这样我们就可以使用(listof *symbol*)表示所有的符号表，使用(listof *number*)表示所有的数表，使用(listof (listof *number*))表示所有的由数表构成的表，等等

在合约中，我们使用(listof *X*)来表示该函数可以使用任何的一种表作为参数：

```
;; length : (listof X) -> number
;; 计算表的长度
(define (length alon)
  (cond
```

```
    [(empty? alon) empty]
    [else (+ (length (rest alon)) 1)]))
```

这里，*X* 只是一个变量，代表某种数据类型的名字。现在，如果把 length 作用在一个元素上，例如(listof *symbol*)或者(listof *IR*)，就能得到一个数。

函数 length 是简单多态的一个例子，它能够作用于各种类型的表上。事实上还有许多其他简单多态的函数，但是更为一般的例子需要我们定义像 *filter1* 这样的函数，使用数据和函数的参数形式作为参数。这两者的结合产生极其强大的功能，极大地方便了软件系统的构建和维护。为了能更好地理解这一点，我们会接着讨论 Scheme 语法的改变以及书写合约的新方法。

习题

习题 19.2.2 使用习题 19.1.6 中抽象的 *sort* 函数，对 *IR* 表进行升序和降序的排列。

习题 19.2.3 *pair*（对）的结构体定义是：

```
(define-struct pair (left right))
```

其参数数据定义是：

> (*pair X Y*) 是结构体：
> (make-pair *l r*)
> 其中 *l* 是 *X* 类型的而 *r* 是 *Y* 类型的。

使用这种抽象数据定义，我们可以描述许多不同的对：

1. (*pair number number*)，由两个数组成的对；
2. (*pair symbol number*)，由符号和数组成的对；
3. (*pair symbol symbol*)，由两个符号组成的对。

当然，在所有的这些范例中，每个对都包含 pair-left 和 pair-right 两个部分。

通过结合表以及对的抽象数据定义，我们可以用如下的一行代码来表示由参数对构成的表：

(listof (*pair X Y*))

这种抽象数据类型的几个实例是：

1. (listof (*pair number number*))，由数对组成的表；
2. (listof (*pair symbol number*))，由符号和数对组成的表；
3. (listof (*pair symbol symbol*))，由符号对组成的表。

对这里的每一个类型，试着举一个实际的例子。

设计函数 *lefts*，该函数读入由(*pair X Y*)组成的表，返回对应的由 *X* 组成的表；换句话说，提取出输入表中每个元素的 *left* 部分。

习题 19.2.4 下面是非空表的参数数据定义：

> (*non-empty-listof ITEM*) 是下列二者之一：
> 1. (cons s empty)。
> 2. (cons *s l*)，其中 *l* 是(*non-empty-listof ITEM*)
> 并且 *s* 总是 *ITEM*。

提示：用一个固定类型的数据代替 *ITEM*，开发出 *last* 函数的草稿。然后再用 *ITEM* 替换回该数据类型。



函数也是值

第 19 章中的函数扩展了我们对计算的认识。读入数和符号的函数相对简单一些；读入结构体和表的函数则稍微复杂一些，但仍然在我们的掌握之内；而以函数作为参数的函数就超出了我们的知识。事实上，第 19 章中的一些函数违背了第 8 章中给出的 Scheme 文法。

在这一章中，我们将讨论如何调整 Scheme 文法和计算规则，从而认识到函数的角色实际上和数据是一样的。如果没有这样的概念，就不可能对函数进行抽象。一旦了解了这些概念，我们就能够学习如何来编写这类函数的合约。本章的最后一节介绍返回函数的函数，这是另外一个强大的抽象工具。

20.1 语法和语义

第 19 章中的函数在两个地方违反了基本的 Scheme 文法。第一，函数和基本操作的名称在调用时被用作参数。虽然参数也是表达式，但是表达式类型中并不包含基本操作和函数的名称。表达式包括变量，而我们原先的标准只允许那些在变量定义中出现的变量被用作函数的参数。第二，参数被像函数一样使用，也就是说，被用在调用者的位置。但是，第 8 章给出的文法只允许函数名和基本操作名出现在这个位置上。

既然问题已经讲清楚了，就需要对文法做出改进。首先，我们应当在<*exp*>的定义中加入函数名和基本操作名。其次，调用的第一个位置应当允许函数名和基本操作以外的东西，至少应该允许作为函数参数的变量，预料到函数的其他用途，我们还应允许表达式出现在该位置上。

这三处改变总结起来就是：

```
<exp>      =   <var>
             | <prm>
             | (<exp> <exp> ...<exp>)
```

图 20.1 给出了完整的 Scheme 文法，包含了所有到目前为止所讨论的扩展。该图表明，有关抽象函数的调整并没有使文法变得更冗长，反而使其变得简单了。

对于运算规则，这一点同样成立。事实上，运算规则并没有改变，只是值的集合发生了改变。为了使函数也能作函数的参数，最简单的改变方式是让值的集合包括函数名以及基本操作名：

```
<val>   =    <boo> | <sym> | <num> | empty | <lst>
           | <var> (已定义的函数的名称)
           | <prm>
<lst>    = empty | (cons <val> <lst>)
```

换句话说，现在如果想判断是否可以对函数实施替代法则，我们必须确保所有的参数都是值，但是此时必须意识到函数名和基本操作名也是一种值。

```
<def>  =  (define (<var> <var> ...<var>) <exp>)
        |(define <var> <exp>)
        |(define-struct <var> (<var> <var> ...<var>))
<exp>  =  <var>
        | <boo>
        | <sym>
        | <prm>
        | empty
        | (<exp> <exp> ...<exp>)
        | (cond (<exp> <exp>) ...(<exp> <exp>))
        | (cond (<exp> <exp>) ...(else <exp>))
        | (local (<def> ...<def>) <exp>)
<var>  =  x | area-of-disk | circumference | ...
<boo>  =  true | false
<sym>  =  'a | 'doll | 'sum | ...
<num>  =  1 | -1 | 3/5 | 1.22 | ...
<prm>  =  + | - | cons | first | rest | ...
```

图 20.1 Intermediate Student Scheme 文法

习题

习题 20.1.1 假设 DrScheme 的 Definitions 窗口中包含(define (*f x*) *x*)，确定下列表达式的值：

```
1. (cons f empty)
2. (f f)
3. (cons f (cons 10 (cons (f 10) empty)))
```

解释为什么它们是值，而其余表达式不是值。

习题 20.1.2 说明为什么下列句子是合法的定义：

```
1. (define (f x) (x 10))
2. (define (f x) f)
3. (define (f x y) (x 'a y 'b))
```

习题 20.1.3 开发函数 *a-function=?*，该函数读入两个把数映射到数的函数，判断这两个函数对于输入、3 以及-5.7 是否返回相同的结果。

我们能否奢望定义出 *function=?*函数，该函数判断两个（把数映射到数的）函数是否完全相同？

20.2 抽象函数和多态函数的合约

在第一次由 *below* 和 *above* 函数抽象出 *filter1* 时，我们并没有给出它的合约。与以前定义的函数不同，*filter1* 使用了一种我们从未用过的值来当参数：基本操作名和函数名。尽管如此，我们最终确定，*filter1* 的第一个参数 *rel-op* 应当是函数，而且这个函数应该读入两个数，返回布尔值。

过去，如果要写出 *rel-op* 函数的合约，一般是：

```
;; rel-op : number number -> boolean
```

鉴于函数和基本操作也是值，这个合约中的->符号描绘了这样一类值：函数和基本操作。->号左边

的名称指定了这个函数运行时所需的参数类型；->号右边的名称指定了这个函数返回值的类型。一般来说，

```
(A B -> C)
```

表示这样的一类函数：它读入一个 *A* 类型的元素和一个 *B* 类型的元素，返回 *C* 类型的元素。即，它们是"把 *A* 和 *B* 映射到 *C*"的函数。

类似于上一章中的(listof...)符号，通过结合其他数据类型，箭头符号可以确定一种数据类型。对 listof 而言，我们用数据定义来确定它们的含义。照着这个样子，其他人也可以定义自己的数据定义缩写。对箭头而言，我们约定它表示函数类型，这种约定对我们很有利。

使用箭头符号，我们可以给出 *filter1* 的合约和用途说明：

```
;; filter1 : (number number -> boolean) lon number -> lon
;; 构造含有 alon 中所有(rel-op n t)为真的 n 的表
(define (filter1 rel-op alon t) ...)
```

这个合约的与众不同之处在于，它说明第一个参数的类型必须不是数据定义中引入的名称，而应当是使用箭头符号直接定义的数据。更具体地说，它指明第一个参数必须是个函数（或者是基本操作），而且还确定了它的参数和返回值的类型。

习题

习题 20.2.1　解释下列函数类型：

```
1. (number -> boolean),
2. (boolean symbol -> boolean),
3. (number number number -> number),
4. (number -> (listof number)),
5. ((listof number) -> boolean)。
```

习题 20.2.2　写出下列函数的合约：

1. *sort*，使用两个参数，第一个参数是数表，第二个参数是函数，该函数使用两个数作参数，生成布尔值；*sort* 的返回值是数表。

2. map，使用两个参数，第一个参数是数到数的函数，第二个参数是数表；map 返回数表。

3. *project*，使用两个参数，第一个参数是由符号表组成的表，第二个参数是符号表到符号的函数；*project* 的返回类型是符号表。

第二个版本的 *filter1* 是 *below* 和 *below-ir* 的抽象，它的合约与前一个 *filter1* 并没有太大的不同，但是从 *below-ir* 的抽象表明，*filter1* 应当不仅能处理数表，还能处理所有类型的表。

我们使用(listof *X*) 表示所有类型的表。先试着写出 *filter1* 的合约：

```
;; filter1 : ... (listof X) number -> (listof X)
```

filter1 之所以能够用于不同类型的表，其关键是使用了一个能够将表中的元素与第二个参数（一个数）进行比较的函数，换句话说，*filter1* 的第一个参数是一个函数：

```
(X number -> boolean)
```

表示它用 *X* 类型的表和数作为参数，产生布尔值。把上述两条放到一起，我们就得到了如下的合约：

```
;; filter1 : (X number -> boolean) (listof X) number -> (listof X)
```

在 length 函数的合约中，X 表示任意一种 Scheme 数据类型。我们可以用任何东西代替这个 *X*，只要三个 *X* 出现的地方都被同一样东西代替。因此，在第一个参数、第二个参数以及返回值位置使用的三个 *X*，表示 *rel-op* 函数的参数是：第一个参数是 *X* 类型的，第二个参数是 *X* 类型的表，*filter1* 的返回值也是 *X* 类型的表。

当调用 *filter1* 时，我们必须保证参数是有意义的。假设我们想要计算

```
(filter1 < (list 3 8 10) 2)
```

在开始计算之前，我们应当先检查合约，确认 *filter1* 可以使用参数<和(list 3 8 10)。快速检查的结果是：因为<属于类型：

```
(number number -> boolean)
```

而(list 3 8 10)属于类型：

```
(listof number)
```

所以它们都是有意义的。

如果我们把 *X* 替换成 ***number***，那么这两者（指合约与实际调用）的类型就完全相同了。更为一般地说，为了保证参数是有意义的，我们必须找到一种合约中变量的替换，使得合约与函数的参数类型相匹配。

接着我们考虑第二个例子：

```
(filter1 <ir LOIR 10)
```

现在，因为$<_{ir}$的合约是：

```
(IR number -> boolean)
```

而 *LOIR* 属于类型(**listof** *IR*)，所以我们必须把 *X* 替换成 *IR*。这样，所有的参数都属于正确的类型，所以该调用仍然是合法的。

我们再来观察另外一个例子：使用 *filter1* 函数从存货记录表中提取出所有有着相同名称的玩具：

```
;; find : (listof IR) symbol -> (listof IR)
(define (find aloir t)
  (filter1 eq-ir? aloir t))

;; eq-ir? : IR symbol -> boolean
(define (eq-ir? ir p)
  (symbol=? (ir-name ir) p))
```

在这个例子中，很容易检查函数是否能够正确工作。我们的任务是理解它是怎样与 *filter1* 的合约匹配的。较为明显的问题是，“限值”参数是符号，而不是数，这就与 *filter1* 的合约产生了矛盾。为了解决这个问题，我们必须给限值引入另一个变量，比方说 ***TH***，代表某种数据类型集合：

```
;; filter1 : (X TH -> boolean) (listof X) TH -> (listof X)
```

现在，我们可以用某种数据类型代替 ***X***，而用另一种（或者是同一种）数据类型代替 ***TH***。特别，通过把 *X* 替换成 ***IR*** 而把 ***TH*** 替换成 ***symbol***，调用

```
(filter1 eq-ir? LOIR 'doll)
```

就可以与 *filter1* 的合约匹配，从而正常运行。

习题

习题 20.2.3 使用 *filter1* 定义一个函数，它读入一个符号表，提取出所有表中与'car 不相同的元素。给出与 *filter1* 相应的合约。

习题 20.2.4 写出下列函数的合约：

1. *sort*，使用两个参数，第一个参数是数表，第二个参数是函数，该函数使用两个数作参数，生成布尔值；*sort* 的返回值是数表。

2. map，使用两个参数，第一个参数是表到 *X* 的函数，第二个参数是表；map 返回 *X* 类型的表。

3. *project*，使用两个参数，第一个参数是由表组成的表，第二个参数是表到 *X* 的函数；*project* 的返回类型是 *X* 类型的表。

与习题 20.2.2 进行比较。

合约与类型：　总而言之，函数的合约是由类型组成的。类型是下列四者之一：

1. 基本类型，例如数、符号、布尔值或者 empty；
2. 已定义的类型，例如 *inventory-record*、*list-of-numbers* 或者 *family-tree*；
3. 函数类型，例如(*number* -> *number*)或者(*boolean* -> *symbol*)；
4. 参数类型，要么是已定义的类型，要么是含有类型变量的函数类型。

如果要使用含有参数类型的函数，我们必须先找到一个（所有函数合约中变量的）替换，使得所有的参数都属于合适的类型。如果做不到这一点，我们要么修改函数的合约，要么认为这个函数不适用于这种情况。



抽象设计的例子



刚开始讨论加法时，我们通过使用具体的例子来学习；稍后，我们学习如何相加任意的两个数，也就是说，我们形成抽象的加法运算概念。再后来，我们直接学习抽象的表达式：计算工资的表达式，换算温度的表达式，或者是计算几何图形面积的表达式。简而言之，我们一开始先学习具体的实例，然后再学习抽象的概念，但是最终，我们可以略过学习具体实例而直接掌握抽象事物。

在这一章中，我们讨论由具体实例形成抽象的设计诀窍。接着，在 21.5 节和第 22 章中，我们学习其他的函数抽象方法。

21.1 从实例中抽象

从实例中构建抽象相当简单，如同我们在第 19 章中看到的，从两个具体的函数开始，比较它们，找出不同点，然后开始抽象。我们把这些步骤组织成诀窍：

比较：如果发现两个函数的定义（除了少数地方以外）基本相同，就比较它们，标记出不同点。如果不同之处只是值，就可以对这两个函数进行抽象。

警告：非值抽象：如果要抽象的东西不是值，诀窍需要有本质的修改。

下面是一对相似函数的定义：

```
;; convertCF : lon -> lon
(define (convertCF alon)
  (cond
    [(empty? alon) empty]
    [else
      (cons (C->f (first alon))
     (convertCF (rest alon)))]))
```

```
;; names : loIR -> los
(define (names aloIR)
  (cond
    [(empty? aloIR) empty]
    [else
      (cons ((IR-name (first aloIR))
     (names (rest aloIR)))]))
```

这两个函数都把一个函数作用于表中的每个元素，它们唯一的差别是作用到表中每个元素上的函数不同。定义中用方框标出了这一差异，它们是两个不同的函数类型的值，所以我们可以进行抽象。

抽象：接下来，我们用同样的名称替换每处不同点，并把这些新的名称加入到参数表中。例如，如果两个函数之间有三处不同，我们就需要三个名称。这样，两个函数的定义就完全一样了。只要把函数的名字也换成新的，我们就得到了这两个函数的抽象。

就上面的例子而言，我们可以得到如下的一对函数：

```
(define (convertCF f alon)
  (cond
    [(empty? alon) empty]
    [else
      (cons (f (first alon))
```

```
(define (names f aloIR)
  (cond
    [(empty? aloIR) empty]
    [else
      (cons (f (first aloIR))
```

```
    (convertCF f (rest alon)))]))          (names f (rest aloIR)))]))
```

这里，我们用 *f* 替换了两个函数的不同处，并且把 *f* 加入到函数的参数表中。现在，我们用一个新的名称代替 *convertCF* 以及 *names*，得到抽象后的函数：

```
(define (map f lon)
  (cond
    [(empty? lon) empty]
    [else (cons (f (first lon))
            (map f (rest lon)))]))
```

按照函数式程序设计语言的惯例，这个新函数的名称为 map。

测试：现在我们必须验证新的函数是否是原来函数的正确抽象。一种好方法是，使用抽象函数反过来定义原来的函数，并且测试这样得到的函数是否与原先的函数等价。

在大多数情况下，用抽象的函数反过来定义原来的函数相当直截了当。假设抽象的函数名称是 *f-abstract*，而原先的某一个函数名称是 *f-original*，而 *f-original* 只有一个参数。如果 *f-original* 与另外一个具体的（抽象前的）函数只有一处不同，我们称这个不同点为 *boxed-value*，那么我们定义如下的函数：

```
(define (f-from-abstract x)
  (f-abstract boxed-value x))
```

对于任何合适的值 *V*，现在(*f-from-abstract V*)都会产生与(*f-original V*)一样的结果。

回到例子中来，下面是两个新的定义：

```
;; convertCF-from-map : lon -> lon
(define (convertCF-from-map alon)
  (map C->F alon))
```

```
;; names-from-map : loIR -> los
(define (names-from-map aloIR)
  (map IR-name aloIR))
```

为了确保这两个定义与原来的函数等价，也就是确保 map 函数是原来的两个函数的正确抽象，我们使用在开发函数 *convertCF* 和 *names* 时用到过的例子来测试这两个新函数。

合约：为了使抽象的函数真正起作用，我们必须给出它的合约。如果在我们的诀窍的第二阶段中，两个函数的不同之处是个函数，那么合约中就会含有箭头。此外，为了得到一个有着广泛用途的合约，我们有时必须使用含有参数的数据定义，并且设计一个参数类型。

以 map 函数为例，一方面，如果我们把 map 看作是 *convertCF* 的抽象，那么合约应当是：

```
;; map : (number -> number) (listof number) -> (listof number)
```

另一方面，如果我们把 map 看作是 *names* 的抽象，那么合约应当是：

```
;; map : (IR -> symbol) (listof IR) -> (listof symbol)
```

但是，第一个合约在第二种情形下毫无意义，反之亦然。为了适应这两种情况，我们必须理解到底 map 能作些什么，然后得出合约。

考虑 map 的定义，我们可以知道 map 把其第一个参数（函数）作用到第二个参数（表）中的每一元素。这意味着作为第一个参数的函数必定使用表中的数据作为输入，也就是说，如果 *lon* 中的数据是 *X* 类型的，那么 *f* 的合约应该是：

```
;; f : X -> ???
```

另外，map 产生的表是由 *f* 作用到表中的每个元素所构成的，所以，如果 *f* 产生的结果是 *Y* 类型的，那么 map 就会产生 *Y* 类型的表。把所有这些翻译成合约所使用的语言，就是：

```
;; map : (X -> Y) (listof X) -> (listof Y)
```

这个合约说明，map 是从 *X* 到 *Y* 的函数以及 *X* 类型的表产生 *Y* 类型的表的函数——不论 *X* 和 *Y* 到底是什么。

一旦完成了对两个（或更多）函数的抽象，我们应当检查这样抽象函数还有没有其他用途。在许多

情况下，抽象函数比我们一开始所想要的函数功能要广泛许多，并且更易于阅读、理解和维护。例如，当需要使用一个函数把一个表（通过映射表中的每一个元素）映射到另一个表时，我们现在就可以使用 map 了。如果那个（映射）函数是一个基本操作，或者是一个现有的函数，我们甚至不需要再去写一个函数，只要写一个表达式就可以完成这项任务。不幸的是，并不存在一个通用的诀窍可以指导我们发现这类功能，我们所能做的只是多实践，并且留心观察哪里适合使用抽象函数。

习题

习题 21.1.1 定义下列两个函数的抽象，并将其命名为 *tabulate*：

```
;; tabulate-sin : number -> lon
;; 列出从 n 到 0（包括 0）的正弦函数表
(define (tabulate-sin n)
  (cond
    [(= n 0) (list (sin 0))]
    [else
      (cons (sin n)
     (tabulate-sin (sub1 n)))]))
```

```
;; tabulate-sqrt : number -> lon
;; 列出从 n 到 0（包括 0）的根号函数表
(define (tabulate-sqrt n)
  (cond
    [(= n 0) (list (sqrt 0))]
    [else
      (cons (sqrt n)
     (tabulate-sqrt (sub1 n)))]))
```

不要忘了使用 *tabulate* 反过来定义这两个函数。再使用 *tabulate* 定义 square 和 tan 的制表函数。*tabulate* 函数正确、通用的合约又是怎样的？

习题 21.1.2 定义下列两个函数的抽象，并将其命名为 *fold*：

```
;; sum : (listof number) -> number
;; 计算 alon 中所有数的总和
(define (sum alon)
  (cond
    [(empty? alon) 0]
    [else (+ (first alon)
        (sum (rest alon)))]))
```

```
;; product : (listof number) -> number
;; 计算 alon 中所有数的乘积
(define (product alon)
  (cond
    [(empty? alon) 1]
    [else (* (first alon)
        (product (rest alon)))]))
```

不要忘了测试你的 *fold* 函数。

完成了 *fold* 的定义和测试后，使用它来定义函数 append。append 的功能是连接两个表，即用第二个表来代替第一个表中的最后那个 empty：

```
(equal? (append (list 1 2 3) (list 4 5 6 7 8))
        (list 1 2 3 4 5 6 7 8))
```

最后，用 *fold* 来定义 map。

比较这四个例子，然后给出 *fold* 的合约。

习题 21.1.3 定义下列两个函数的抽象，并将其命名为 *natural-f*：

```
;; copy : N X -> (listof X)
;; 生成一个表，表中含有 n 个 obj
(define (copy n obj)
  (cond
    [(zero? n) empty]
    [else (cons obj
                (copy (sub1 n) obj))]))
```

```
;; n-adder : N number -> number
;; 使用每次加一
;; 的方法把 n 加到 x 上
(define (n-adder n x)
  (cond
    [(zero? n) x]
    [else (+ 1
             (n-adder (sub1 n) x))]))
```

不要忘了测试 *natural-f*。然后使用 *natural-f* 来定义函数 *n-multiplier*。*n-multiplier* 读入 *n* 和 *x*，只使用加法，返回 *n* 乘以 *x*。通过这个例子，总结出 *n-multiplier* 的合约。

提示：这两个函数的差别比习题 21.1.2 中函数 *sum* 和 *product* 的差别大，特别是两个函数的基本情况并不相同，在一个函数中，基本情况是函数的参数，而在另一个函数中，基本情况是一个常量。

明确表达一般化的合约：为了增加抽象函数的用途，我们必须给出其最为通用的合约。原则上，抽象合约的诀窍与抽象函数的诀窍是一致的，先比较原先的合约，然后用变量来代替不同之处。但是，这个过程相当复杂，并且需要大量实践才能掌握。

我们首先从 *convertCF* 和 *names* 的例子开始：

```
(listof number)    ->    (listof number)
(listof IR)     ->      (listof symbol)
```

比较这两个合约，可知它们有两点不同，在→的左边，*number* 与 *IR* 不同；在→的右边，*number* 与 *symbol* 不同。

考虑抽象诀窍的第二阶段，最为自然的合约是：

```
(number -> number)     (listof number)    ->    (listof number)
(IR -> symbol)          (listof IR)         ->     (listof symbol)
```

这个新的合约暗示着这样一种模式：第一个参数（函数）使用第二个参数(表)的每一个元素作为输入，并且它的输出构成了 map 函数的输出。如果我们把 *IR* 和 *symbol* 替换成变量，我们就得到了抽象的合约，而且这就是 map 函数真正的合约：

```
map : (X -> Y) (listof X) -> (listof Y)
```

作为检查，只要把 *X* 替换成 *number*，把 *Y* 替换成 *number*，我们就能得到前一个合约。

下面是另外一组例子：

```
number      (listof number)       ->      (listof number)
Number      (listof IR)           ->      (listof IR)
```

它们分别是 *below* 和 *below-ir* 的合约。这两个合约在两个地方有所不同：作为输入和输出的**表**不同。通常，在抽象的第二阶段中，函数还使用到另一个参数：

```
(number number-> boolean)     number     (listof number) ->    (listof number)
(number IR -> boolean)        number     (listof IR)     ->    (listof IR)
```

新的参数是个函数，在前一个合约中，该函数的参数是数，而在后一个合约中，该函数的参数是 *IR*。

比较这两个合约，可知 *number* 和 *IR* 是对应的不同点，所以我们用一个变量替换掉它们，从而得到两个相同的合约：

```
(number X -> boolean)     number     (listof X) ->  (listof X)
(number X -> boolean)     number     (listof X) ->  (listof X)
```

更为仔细地研究 *filter1* 的定义，我们发现第二个参数必然与第一个参数（函数）的第一个参数相同，所以还可以把 *number* 替换成 *Y*。

新的合约就是：

```
filter1 : (Y X -> boolean) Y (listof X) -> (listof X)
```

因为第一个参数（函数）产生的结果用作判断条件，所以它必定是 *boolean*，因此我们已经找到了最为一**般化**的合约。

通过这两个例子，我们示范了如何给出一**般化**的合约。首先比较抽象前函数的合约，通过逐一替换

对应位置上的不同类型，逐渐地得到一般化的合约。为了保证这样得到的合约能够正常工作，我们还要检查该合约是否能够正确描述特定的应用。

21.2 抽象表处理函数的练习

Scheme 提供了大量处理表的抽象函数，图 21.1 列出了其中最主要的函数的说明。使用这些函数可以极大地简化程序设计过程，并且方便人们更快地阅读程序。本节中的习题给你一个了解这些函数的机会。

```
;; build-list : N (N -> X) -> (listof X)
;; 构造(list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

;; filter : (X -> boolean) (listof X) -> (listof X)
;; 用 alox 中所有满足 p 的元素构造一个表
(define (filter p alox) ...)

;; quick-sort : (X X -> boolean) (listof X) -> (listof X)
;; 按照 cmp 的顺序给 alox 表排序
(define (quick-sort cmp alox) ...)

;; map : (X -> Y) (listof X) -> (listof Y)
;; 通过把 f 作用到 alox 中的每个元素，构造一个表
;; 也就是说，(map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f alox) ...)

;; andmap : (X -> boolean) (listof X) -> boolean
;; 检测 p 是否对于 alox 中的每个元素都成立
;; 也就是说，(andmap p (list x-1 ... x-n)) = (and (p x-1) (and ... (p x-n)))
(define (andmap p alox) ...)

;; ormap : (X -> boolean) (listof X) -> boolean
;; 检测 p 是否对于 alox 中的至少一个元素成立
;; 也就是说，(ormap p (list x-1 ... x-n)) = (or (p x-1) (or ... (p x-n)))
(define (ormap p alox) ...)

;; foldr : (X Y -> Y) Y (listof X) -> Y
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

;; foldl : (X Y -> Y) Y (listof X) -> Y
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)

;; assf : (X -> boolean) (listof (list X Y)) -> (list X Y) or false
;; 找出 alop 中使 p 成立的第一个元素
(define (assf p? alop) ...)
```

图 21.1 Scheme 中内建的表处理函数

习题

习题 21.2.1 使用 build-list

1. 创建表(list 0 ... 3)以及(list 1 ... 4);
2. 创建表(list .1 .01 .001 .0001);
3. 定义 *evens*,它读入自然数 *n*,返回由前 *n* 个偶数组成的表;
4. 定义习题 21.1.1 中的 *tabulate*;
5. 定义 *diagonal*,它读入自然数 *n*,创建由 0、1 表组成的表(对角矩阵)。
   例子:

```
(equal? (diagonal 3)
        (list
         (list 1 0 0)
         (list 0 1 0)
         (list 0 0 1)))
```

在函数定义的过程中,如果需要使用辅助函数,请使用局部函数。

习题 21.2.2 使用 map 定义下列函数:

1. *convert-euro*,基于 1.22 欧元合一美元的汇率,把美元数额的表转换成欧元数额的表;
2. *convertFC*,把含有华氏温度的表转换成含有摄氏温度的表;
3. *move-all*,输入 *posn* 结构体的表,在每个 *x* 成分上加上 3。

习题 21.2.3 下面是 DrScheme 提供的 *filter* 函数:

```
;; filter : (X -> boolean) (listof X) -> (listof X)
;; 用 alon 中所有使 predicate?成立的元素构造一个 X 类型的表
(define (filter predicate? alon)
  (cond
    [(empty? alon) empty]
    [else (cond
        [(predicate? (first alon))
         (cons (first alon) (filter predicate? (rest alon)))]
        [else (filter predicate? (rest alon))])]))
```

使用 filter 定义下列函数:

1. *eliminate-exp*,读入一个数 *ua* 以及 *toy* 结构体的表(*toy* 结构体包含 *name* 和 *price*),返回包含所有 *price* 小于 *ua* 的元素组成的表;

2. *recall*,读入玩具的名字 *ty* 以及名字表 *lon*,返回除了 *ty* 以外 *lon* 中所有的元素;

3. *selection*,读入两个名字表,选出所有在两个表中都出现的名字(也就是说,从第二个表中找出在第一个表中出现的名字)。

21.3 抽象与惟一控制点

与编辑论文一样,抽象程序有许多好处:创建抽象通常可以简化其他的定义;抽象的过程可能揭示出现有函数的问题。但是,抽象最重要的好处是,它为程序的功能性提供了一个惟一控制点,换句话说,它把所有和某个特定任务相关的定义都(尽可能地)集中到一起。

把与某个任务相关的定义其中在一起使得程序更易于维护。大致说来,程序的维护包括:修改程序,

使之能在以前未测试过的情形下正常运行；扩展程序，使之能处理新的或者以前未预见到的问题；将某些信息表达修改成数据（例如历法日期）。如果我们把所有的东西都放在一个地方，修改程序时就只要修改一个函数，而不是四个或五个相似的程序；扩展程序时也只要扩展一个函数，无须涉及相关的函数；而改变数据表示法意味着改变一个一般化的数据遍历函数，而不必修改所有源与同一模板的函数。概括说来：

> **创建抽象的原则**
>
> *建立抽象，而不是反复复制和修改同一段程序。*

经验告诉我们，维护程序的代价非常大。通过正确组织程序的结构，程序设计者可以把维护的代价降到最小。组织函数结构的第一个**原则**是：函数的结构与输入数据的结构要相配。如果所有的程序设计者都按照这个**原则**编写程序，当一些输入数据类型改变时，我们可以很容易地修改和维护函数。第二个**原则**是：引入合适的**抽象**。一个**抽象**函数至少为两个（通常更多）不同的函数创建了一个惟一控制点，引入了抽象之后，我们通常可以找到新函数的更多用途。

创建抽象最基本的工具就是我们的设计诀窍。需要练习才能掌握它。在练习的过程中，我们增强了建立和使用抽象的能力。好的程序设计者总是积极地建立新的抽象，把相关的任务放入惟一控制点。这里我们通过函数抽象学习这个过程。虽然并不是所有的语言都像 Scheme 一样提供方便的抽象函数，但是现代程序设计语言通常支持类似的概念，所以说学习 Scheme 是学习其他语言的一个最好准备[1]。

21.4　补充练习：再论图片移动

在第 6.6 节、7.7 节和 10.3 节中，我们已经学习过如何在画布上移动图片。这个问题包括两个部分：移动单独的图形与移动图形的表，也就是图片。对于前者，我们使用绘制、清除和平移函数；对于后者，我们使用能够绘制、清除和平移整个表的函数。即使是草草地看一眼这些函数，我们也发现其中存在着许多重复。下面的习题通过手工抽象以及使用 Scheme 的内建函数消除这些重复。

习题

习题 21.4.1　把函数 *draw-a-circle* 和 *clear-a-circle* 抽象成函数 *process-circle*，再使用 *process-circle* 定义 *translate-circle*。提示：如果原来的某个函数并不能很好地直接抽象，我们可以定义辅助函数。这里，请使用 define。第 24 章介绍了一种方便的、重要的定义辅助函数的方法。

习题 21.4.2　把函数 *draw-a-rectangle* 和 *clear-a-rectangle* 抽象成函数 *process-rectangle*，再使用 *process-rectangle* 定义 *translate-rectangle*。

习题 21.4.3　把函数 *draw-shape* 和 *clear-shape* 抽象成函数 *process-shape*，再使用 *process-shape* 定义 *translate-shape*。比较 *process-shape* 与模板 *fun-for-shape* 的异同。

习题 21.4.4　使用 Scheme 提供的 *map* 与 *andmap* 函数定义 *draw-losh*、*clear-losh* 和 *translate-losh*。

习题 21.4.5　修改习题 21.4.3 和习题 21.4.4 中的函数，使得图片能够在画布上上下移动。

修改所有的定义，使得图形包括直线；直线包含起点、终点和颜色。

定义 LUNAR（一个形状表），描绘登月舱的草图（参见图 21.2），该表应当由矩形、圆和直线组成。

---

[1] 在基于类的面对对象程序设计语言中，近来一种流行的抽象方法是继承。继承与函数的抽象很类型，不过它更强调（对象之间）改变的方面，而忽略不变的方面。



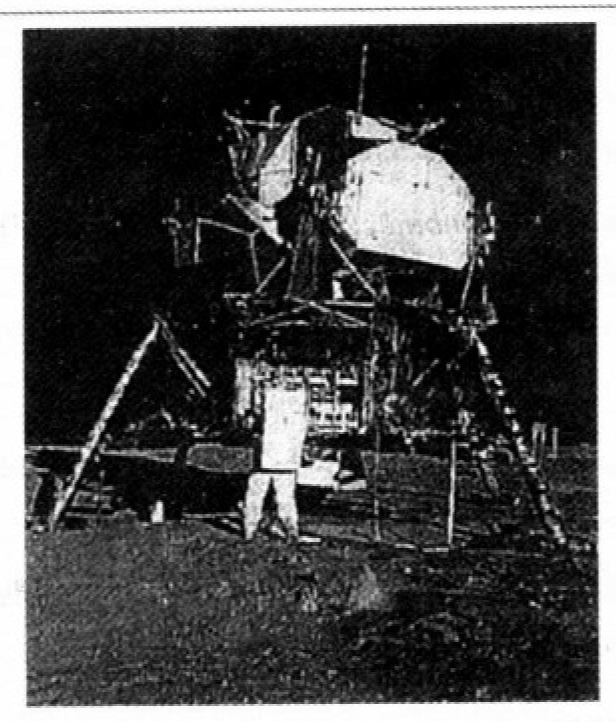


图 21.2 阿波罗 11 号登月舱国家航空和航天局（NASA）：国家航天科学数据中心

设计程序 *lunar-lander*，它能把 LUNAR 放在画布顶端，然后使用刚才修改得到的函数上下移动登月舱。

使用教学软件包中的 arrow.ss，让用户来控制登月舱的何时移动以及移动的速度是多少：

```
(start 500 100)
(draw LUNAR)
(control-up-down LUNAR 10 move-lander draw-losh)
```

时间允许的话，修改函数，使得用户可以让登月舱上、下、左、右移动。使用 arrow.ss 中的 *controller* 函数控制移动的方向。

21.5 注意：由模板设计抽象

在本部分的一开始，我们就讨论过如何由模板设计函数。更精确地说，在设计一组有着同样输入类型的函数时，我们反复使用同样的模板。所以毫不令人奇怪，这些函数的定义看上去都很类似，而且我们将来会对它们进行抽象。

实际上，我们可以直接从模板抽象。尽管这是一个很复杂的话题，甚至还是一个程序设计语言研究的热点，但是我们可以用一个简短的例子来论述这个问题。考虑表的模板：

```
(define (fun-for-l l)
  (cond
    [(empty? l) ...]
    [else ... (first l) ... (fun-for-l (rest l)) ...]))
```

该模板有两个空缺，每个子句各有一个。如果要定义一个表处理函数，只需填写这些空缺即可。在第一个子句中，一般填上一个普通的值，在第二个子句中，一般把(first *l*)与(*f* (rest *l*))结合起来，其中 *f* 是递归函数。

我们可以用如下的函授对这个程序设计任务进行抽象：

```
;; reduce : X (X  Y -> Y) (listof Y) -> Y
(define (reduce base combine l)
  (cond
```

```
    [(empty? l) base]
    [else (combine (first l)
        (reduce base combine (rest l)))]))
```

它使用了两个额外的参数：*base* 和 *combine*。*base* 是填入第一个空白处的基本成分，*combine* 是负责把第二个子句中的值结合起来的函数。

使用 *reduce*，我们可以定义许多简单的表处理函数，包括图 21.1 中的几乎所有函数。下面就是其中的两个函数：

```
;; sum : (listof number) -> number
(define (sum l) (reduce 0 + l))
```

```
;; product : (listof number) -> number
(define (product l) (reduce 1 * l))
```

对 *sum* 函数来说，其基本成分总是 0；结合第二个子句中两个值的方法是，把表中的第一个元素与自然递归的结果加起来。对 *product* 的解释与此类似。

如果要用 *reduce* 来定义 *sort*，需要先定义一个辅助函数：

```
;; sort : (listof number) -> (listof number)
(define (sort a-list)
  (local ((define (insert an alon)
            (cond
              [(empty? alon) (list an)]
              [else (cond
                  [(> an (first alon)) (cons an alon)]
                  [else (cons (first alon) (insert an (rest alon)))])])))
    (reduce empty insert a-list)))
```

其他的表处理函数也能用类似的方法定义。



使用函数进行抽象设计



我们已经知道了可以以函数作为函数的参数，还知道了创建函数的唯一控制点的重要性。但是函数不仅可以读入函数，也可以返回函数。更精确地说，在新的 Scheme 中，表达式的计算结果可以是函数。因为函数定义的主体部分就是一个表达式，所以函数的输出也可以是函数。在这一章中，我们先讨论这个惊人的概念，然后说明在函数抽象以及相关领域它是多么的有用。

22.1 返回函数的函数

尽管返回函数的函数看起来很奇怪，但事实上它却相当有用。在讨论这个有用的想法之前，我们先来探索函数是如何产生函数的，下面是三个例子：

```
(define (f x) first)
(define (g x) f)
(define (h x)
  (cond
    ((empty? x) f)
    ((cons? x) g)))
```

f 的主体是 first，即一个基本操作，所以无论把 *f* 应用到什么参数上，计算的结果总是 first。类似地，*g* 的主体是 *f*，所以无论把 *g* 应用到任何参数上，计算的结果总是 *f*。最后，取决于我们提供给 *h* 的参数是哪种类型的表，它返回的结果或者是 *f* 或者是 *g*。

虽然这三个例子都没有什么意义，但是它们说明了基本概念。在前两个例子中，函数的主体就是一个函数；在最后一个例子中，函数的主体将计算出一个函数。这三个函数的返回值都不包含输入参数，所以它们没有多大用处。如果我们想要定义包含输入参数、返回函数的函数 *f*，*f* 必须自己定义一个函数，并将它最为结果返回。换句话说，*f* 的主体必须是一个 local 表达式。

回忆一下什么是 local 表达式：local 表达式把定义聚集在一起，要求 DrScheme 使用这些定义作为环境计算表达式的值。local 表达式可以出现在任何一个表达式可以出现的位置，这意味着如下的定义是合法的：

```
(define (add x)
  (local ((define (x-adder y) (+ x y)))
    x-adder))
```

函数 *add* 读入一个数；毕竟，*x* 会被加到 *y* 之上。接着，它用 local 表达式定义了 *x-adder*。因为该 local 表达式的主体就是 *x-adder*，所以 *add* 的返回值就是 *x-adder*。

为了更好地理解 *add*，我们来观察把 *add* 作用于一个数的计算过程：

```
(define f (add 5))
```

```
= (define f (local ((define (x-adder y) (+ 5 y)))
              x-adder))
= (define f (local ((define (x-adder5 y) (+ 5 y)))
             x-adder5))
= (define (x-adder5 y) (+ 5 y))
  (define f x-adder5)
```

最后一步把函数 *x-adder5* 添加到定义之中；在计算过程中，local 表达式的主体——*x-adder5*——是函数名，因而也是值。现在，*f* 已经被定义了，接着我们就可以使用它了：

```
  (f 10)
= (x-adder5 10)
= (+ 5 10)
= 15
```

这就是说，*f* 代表 *x-adder5*——一个函数，它把参数加 5。

使用这个例子，我们可以给出 *add* 的合约和用途说明：

```
;; add : number -> (number -> number)
;; 创建一个函数，把 x 加到其参数上
(define (add x)
  (local ((define (x-adder y) (+ x y)))
    x-adder))
```

add 最为有趣的性质是，它能够"记住" *x* 的值。例如，我们使用 *add* 定义了 *f*，每次调用 *f*，它都使用 5 作为 *x* 的值。这种"记忆"能力正是定义抽象函数的简单诀窍之关键所在，我们会在下一节中讨论这一点。

22.2 把函数当成值来进行抽象设计

通过结合 local 表达式以及把函数当成值，可以简化定义抽象函数的诀窍。回过头来考虑图 19.2 中的第一个例子，如果用 *rel-op* 代替两个函数的不同处，所得的抽象函数就含有一个自由变量。有两种方法可以避免这种情况发生，一种方法是把 *rel-op* 放到抽象函数的参数表中，另一种方法是在定义外加上 local 语句，并把一个以 *rel-op* 为参数的函数放在其前部。图 22.1 就是对 filter 使用第二种方法抽象的结果。如果把 local 语句局部定义的函数作为整个函数的返回值，我们就得到了原来两个函数的抽象。

```
(define (filter2 rel-op)
  (local ((define (abs-fun alon t)
            (cond
              [(empty? alon) empty]
              [else
                (cond
                  [(rel-op (first alon) t)
                   (cons (first alon)
                     (abs-fun (rest alon) t))]
                  [else
                     (abs-fun (rest alon) t)])])))
    abs-fun))
```

图 22.1 使用 local 进行抽象

换一种说法，我们继续使用上一节中 *add* 的例子。与 *add* 一样，*filter2* 需要两个参数，然后定义一个函数，并且返回该函数。如同下面的计算过程所显示的，其返回值能够永久地记住 *rel-op* 参数：

```
  (filter2 <)
= (local ((define (abs-fun alon t)
            (cond
              [(empty? alon) empty]
              [else
                (cond
                       [(< (first alon) t)
                        (cons (first alon)
                           (abs-fun (rest alon) t))]
                       [else
                           (abs-fun (rest alon) t)])])))
    abs-fun)
= (define (below3 alon t)
    (cond
      [(empty? alon) empty]
      [else
     (cond
       [(< (first alon) t)
        (cons (first alon)
          (below3 (rest alon) t))]
       [else
         (below3 (rest alon) t)])]))
   below3
```

请记住，在计算的过程中，当我们把一个 local 函数定义提到最上层的定义之外时，我们必须重命名该函数，因为将来，同一个 local 定义还会再次被计算。在这里，我们用 *below3* 来表示该函数，而实际上，*below* 和 *below3* 唯一的不同就是函数名不同。

根据上述计算，我们可以给(*filter2* <)的结果起个名字，并像 *below* 一样使用它。即：

```
  (define below2 (filter2 <))
```

等价于

```
  (define (below3 alon t)
    (cond
      [(empty? alon) empty]
      [else
     (cond
       [(< (first alon) t)
        (cons (first alon)
          (below3 (rest alon) t))]
       [else
         (below3 (rest alon) t)])]))
  (define below2 below3)
```

这表明 *below2* 仅仅是 *below3* 的另一种叫法，还直接证明了我们的抽象函数正确实现了 *below* 的功能。

这个例子同时也表明另一种抽象的诀窍（不同与第 21 章中给出的诀窍）：

比较：　新的抽象诀窍仍然需要比较两个函数，并且标记出不同点。

抽象：　新方法主要关注定义抽象函数的方法。把（某一个）未抽象函数放入 local 表达式，用该函数的名字作 local 的主体：

```
(local ((define (concrete-fun x y z)
          ... op1 ... op2 ...))
  concrete-fun)
```

然后，把不同之处的名字列成参数表，我们就建立了抽象函数：

```
(define (abs-fun op1 op2)
  (local ((define (concrete-fun x y z)
            ... op1 ... op2 ...))
    concrete-fun))
```

如果 *op1* 或者 *op2* 是个特殊的符号，比如<，我们就在新的环境中给它起一个更有意义的名称。

测试：　为了测试抽象函数，我们仍然使用抽象的函数反过来定义原来的函数。考虑 *below* 和 *above* 的例子，从 *filter2* 中得出 *below* 和 *above* 非常容易：

```
(define below2 (filter2 <))
(define above2 (filter2 >))
```

只要把 *filter2* 函数作用于原先具体函数中不同之处的东西，就可以得到原来的函数了。

合约：　因为抽象函数的输出是一个函数，所以为了描述这种关系，其合约中含有两个箭头，一个箭头的右边含有另一个箭头。

```
filter2 的合约是：
;; filter2 : (X Y -> boolean) -> (Y (listof X) -> (listof X))
```

它读入一个比较函数，返回一个具体的过滤函数。

合约的一般化过程与第 21 章中描述的方法是一样的。

我们已经讨论过使用第一种诀窍进行抽象了，使用第二种诀窍进行抽象就留作习题。

习题

习题 22.2.1　用新的抽象诀窍，定义 21.1 节中 *convertCF* 和 *names* 的抽象。

习题 22.2.2　用新的抽象诀窍，定义 19.1.6 节中 *sort* 的抽象。

习题 22.2.3　用新的抽象诀窍，定义 *fold* 的抽象。回忆一下，*fold* 是从如下两个函数抽象得到的：

```
;; sum : (listof number) -> number
;; 计算 alon 中数的总和
(define (sum alon)
  (cond
    [(empty? alon) 0]
    [else (+ (first alon)
             (sum (rest alon)))]))
```

```
;; product : (listof number) ->
number
;; 计算 alon 中数的乘积
(define (product alon)
  (cond
    [(empty? alon) 1]
    [else (* (first alon)
             (product (rest alon)))]))
```

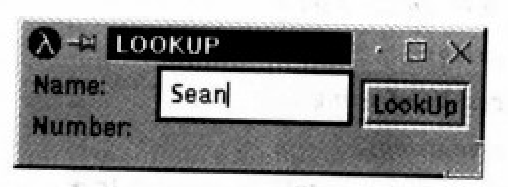
22.3　图形用户界面初探

函数在图形用户界面的设计中起着关键的作用。“界面”指的是程序与用户之间的分界线。如果我

们是唯一的用户，我们当然可以将函数直接作用于 DrScheme 的 Interactions 窗口中的数据。但是，如果其他人想要使用我们的程序，我们必须提供一种方法，让根本不懂任何程序设计知识的人也可以与程序交互。程序与用户交互的方法就叫做用户界面。

对于一般的用户来说，图形用户界面（GUI）是最方便的界面。图形用户界面是一个含有 GUI 对象的窗口。某些 GUI 对象允许用户输入文本；某些 GUI 对象帮助用户调用特定的函数；某些 GUI 对象负责显示函数返回的结果。例如：按钮，用户在按钮上单击鼠标，可以触发一个函数调用；选择菜单，在选择菜单中，用户可以从一些值中选择一个；文本框，用户可以在文本框中输入任意的文本；消息框，程序可以在其中显示文本。





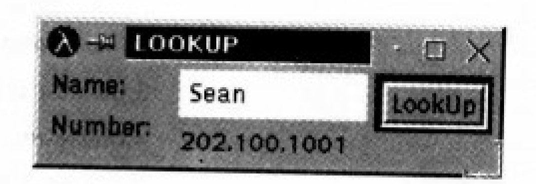


图 22.2　查寻电话号码的简单 GUI

观察图 22.2 中的简单 GUI。左图显示的是其初始状态，其中，GUI 含有一个文本框“Name（人名）”、一个消息框“Number（号码）”和一个“LookUp（查找）”按钮。在中间的图上，用户输入了人名“Sean”，但还没有去按“LookUp”按钮[1]。最后，右图显示了用户按“LookUp”按钮后 GUI 显示出“Sean”的电话号码。

这个程序的核心是从表中查找某人的电话号码的函数。在本书的第二部分中，我们给出了多个这样的函数，不过，它们都使用 DrScheme 的 Interactions 窗口。如果使用图 22.2 中的 GUI，人们无须知道任何有关 Scheme 的知识，照样可以使用我们的函数。

要建立一个图形用户界面，我们先建立与 GUI 对象相应的结构体[2]，然后把它们移交给 GUI 管理器。GUI 管理器使用这些结构体建造可见的窗口。在这些结构体中，有的字段描述 GUI 对象的可见属性，例如按钮的标签、消息框的初始内容以及菜单的各种选项；有的字段代表函数，它们被称为回调函数，因为当用户操作对应的 GUI 对象时，GUI 管理器会调用这些函数。在调用时，回调函数从 GUI 对象处得到字符串以及（自然）数（作为参数），然后调用合适的函数。被调用的函数计算出结果，然后回调函数再把这些结果放入 GUI 对象中，就如同绘图函数在画布上绘制图形。

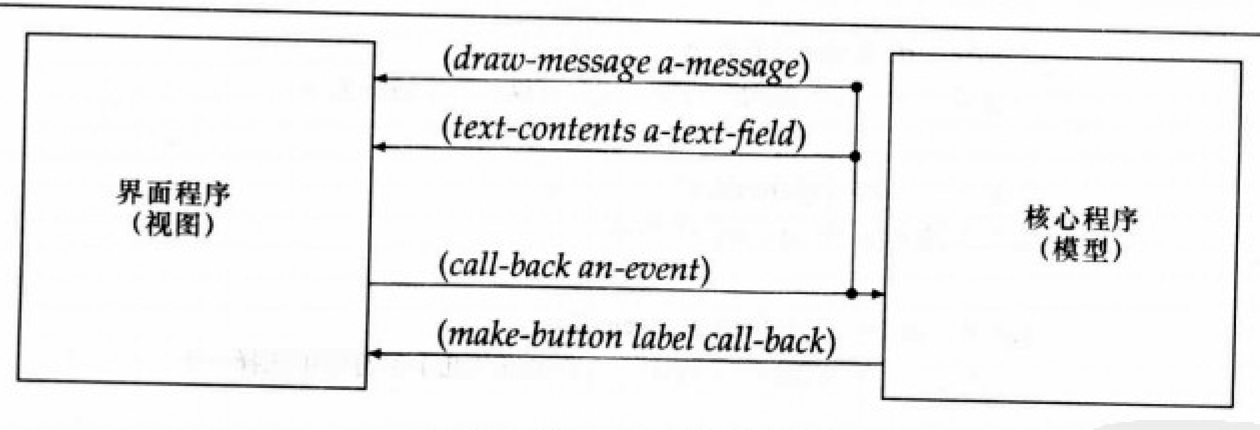


图 22.3　模型—视图体系结构图

理想的程序应该由完全分离的两个部分组成：模型和视图。模型就是我们所学习设计的程序，而视图是负责显示信息、响应用户对鼠标和键盘操作的 GUI 程序。在模型和视图之间的桥梁是控制表达式，图 22.3 形象地描述了这个被称为模型—视图—控制的体系。在该图中，最下层的箭头说明程序是如何用回调函数编排按钮的；从左向右的箭头描述单击鼠标事件以及它是怎样触发回调函数调用的；接着，回调函数使用其他的 GUI 函数获取用户的输入，或者显示核心函数的返回值。

---

[1] 为了避免误解，程序还清空了返回结果的字段。此外，用户也可以直接按回车键，而不必单击按钮，同样可以进行查找。这里我们忽略这些细微之处。

[2] 更精确地说，我们建立对象，不过这里我们无需理解结构体和对象之间的区别。

把程序分成两个部分意味着模型的定义中并不涉及任何与视图有关的内容，视图的定义也不包含任何模型中的数据和函数。这样的结构是花了整整二十年的时间，通过许多成功和失败的经验，逐渐发展起来的。它的优点是，只需调整连接两者的表达式，我们就能使同一个程序使用不同的GUI，或者让同一个GUI为不同的程序服务。另外，构造视图所需的工具与构造模型所使用的工具并不相同。构造视图是劳动密集型的工作，通常涉及到图形设计，万幸的是，通常大部分的视图设计都可以自动完成。而，构建模型总需要进行大量的程序设计工作。

```
gui-item（GUI 对象）是下列之一：
    1.(make-button string (X -> true))
    2.(make-text string)
    3.(make-choices (listof string))
    4.(make-message string)。
    ;; create-window : (listof (listof gui-item)) -> true
    ;; 把 gui-item 加入到窗口（Window）之中，并显示窗口
    ;; 每个 gui-item 的表定义了窗口中的一行 gui-item

    ;; hide-window : -> true
    ;; 隐藏窗口

    ;; make-button : string (event% -> true) -> gui-item
    ;; 创建一个带标签的按钮（button），并调用回调函数

    ;; make-message : string -> gui-item
    ;; 创建一个用来显示消息（message）的对象

    ;; draw-message : gui-item[message%] string -> true
    ;; 在消息对象中显示一条消息
    ;; 该函数将会把现有的消息删除

    ;; make-text : string -> gui-item
    ;; 创建一个（带标签的）对象，用户可以在其中输入文本

    ;; text-contents : gui-item[text%] -> string
    ;; 返回文本框（text）中的字符串

    ;; make-choice : (listof string) -> gui-item
    ;; 创建一个选择菜单（choice），用户可以从几个字符串中选择一个

    ;; get-choice : gui-item[choice%] -> num
    ;; 测定在选择菜单中哪个选项被选中了
    ;; 返回值是选择菜单中从 0 开始的数
```

图 22.4 gui.ss 中定义的操作

现在让我们来学习简化了的GUI：教学软件包gui.ss。图22.4列出了该教学软件包提供的操作[1]。函数 *create-window* 代表GUI管理器，其合约和用途说明很有教育意义。它们说明我们使用一个表来创建窗口，该函数根据可见窗口的行数来安排这个表，每个 *gui-item* 表指定一行。图22.4中给出了以下四种 *gui-item* 的数据定义：

[1] gui-item 不是真正的结构体，它解释了操作的名称的前面部分。

文本框，用(*make-text a-string*)函数创建，允许用户在其中输入任意的文本；

按钮，用(*make-button a-string a-function*)创建，在用户单击时调用一个函数；

选择菜单，用(*make-choice a-list-of-strings*)创建，允许用户从一组选项中选择一个；

消息框，用(*make-message a-string*)创建，允许模型通知用户计算的结果。

配合按钮的函数有一个参数：一个事件。在行多数情况下，我们可以忽略这个事件；它只代表用户单击鼠标动作的信息。

要理解所有这些函数是如何工作的，最好的方法是通过举例来说明。我们的第一个例子是一个规范的 GUI 程序：

```
(create-window (list (list (make-button "Close" hide-window))))
```

它创建只含有一个按钮的窗口，并给该按钮准备了最简单的回调函数：hide-window。该函数的功能是隐藏窗口。只要用户单击“Close”按钮，窗口就会消失。

第二个 GUI 的例子把用户输入到文本框中的文本显示在消息框中。我们先创建文本框和消息框：

```
(define a-text-field
  (make-text "Enter Text:"))
(define a-message
  (make-message " 'Hello World'  is a silly program."))
```

接着我们就可以在回调函数中引用这两者：

```
;; echo-message : X -> true
;; 把文本框中的文本显示到消息框中
(define (echo-message e)
  (draw-message a-message (text-contents a-text-field)))
```

这个回调函数的定义是基于 *gui-item* 的（领域）知识的。具体来说，函数 *echo-message* 获取文本框中的当前内容，存放在字符串 *text-contents* 中，然后使用 *draw-message* 函数把字符串中的内容放入消息框中。为了实现这个功能，我们创建一个窗口，其中有两行对象：

```
(create-window
(list (list a-text-field a-message)
      (list (make-button "Copy Now" echo-message))))
```

第一行中包含了文本框和消息框，第二行中包含一个按钮，按钮的标签是"Copy Now"，对应的回调函数是 *echo-message*。现在，用户可以在文本框中输入文本，单击按钮，然后观察这些文本是否在消息框中出现。

第三个例子，也是最后一个例子的目的是创建含有一个选择菜单、一个消息框和一个按钮的窗口，单击按钮可以把当前选中的内容放入消息框。与上一个例子一样，我们先定义输入和输出的 *gui-item*：

```
(define THE-CHOICES
  (list "green" "red" "yellow"))
(define a-choice
  (make-choice THE-CHOICES))
(define a-message
  (make-message (first THE-CHOICES)))
```

因为在程序中，不止一处要使用到选项表，所以我们用一个单独的变量来定义它。

与上一个例子一样，与按钮相关的回调函数与 *a-choice* 和 *a-message* 交互：

```
;; echo-choice : X -> true
;; 查明 a-choice 当前的选择，
;; 并在 a-message 中显示相应的字符串
```

```
(define (echo-choice e)
  (draw-message a-message
              (list-ref THE-CHOICES
                   (choice-index a-choice))))
```

在这里，回调函数查明用户当前的选择，以（从 0 开始的）数的形式存放在 *choice-index* 中，然后使用 Scheme 提供的 list-ref 函数从 *THE-CHOICES* 中提取出对应的字符串，最后把结果放入消息框中。创建窗口时，我们把 *a-choice* 和 *a-message* 安排在一行中，把按钮安排在另一行中：

```
(create-window
 (list (list a-choice a-message)
       (list (make-button "Confirm Choice" echo-choice))))
```

```
;; 模型:
;; build-number : (listof digit) -> number
;; 把数字表翻译成数
;; 例如: (build-number (list 1 2 3)) = 123
(define (build-number x) ...)

;; ________________________________________

;; 视图:
;; 十个数字的字符串
(define DIGITS
  (build-list 10  number-> string))

;; 由三个数字选择菜单组成的表
(define digit-choosers
  (local ((define (builder i) (make-choice DIGITS)))
    (build-list 3 builder)))

;; 消息框，初始状态显示"Welcome"，被用来显示求出的数
(define a-msg
  (make-message "Welcome"))

;; ________________________________________

;; 控制器:
;; check-call-back : X -> true
;; 获得当前的数字选择，将其转化为一个数，
;; 并以字符串的形式在消息框中显示这个数
(define (check-call-back b)
  (draw-message a-msg
                ( number-> string
                 (build-number
                   (map choice-index digit-choosers)))))
(create-window
 (list
  (append digit-choosers (list a-msg))
  (list (make-button "Check Guess" check-call-back))))
```

图 22.5 把数位显示成数的 GUI

现在我们已经学习了一些基本的 GUI 程序，接下来可以研究一个有着完整核心和 GUI 成分的程序。观察图 22.5 中的定义，该程序的目的是把若干个数字的选择组成一个数，并在消息框中显示这个数。模型部分由函数 ***build-number*** 组成，该函数我们已经多次提到过了，所以图中只给出了其简单说明。程序的 GUI 成分设立三个选择菜单、一个消息框以及一个按钮。控制部分包括一个简单的回调函数，当用户单击唯一一个按钮时，系统调用该回调函数，它读出当前选择菜单中的内容，并把它们移交给 ***build-number***，最后把返回的结果以字符串的形式显示在消息框中。

我们来更加仔细地研究一下回调函数的结构。它由如下的三类函数组成：

1. 最内层的函数读出 *gui-item* 的当前状态，也就是用户的输入。使用现有的函数，我们既可以读出用户在文本框中输入的字符串，也可以从选择菜单中读出当前被选中的那一项。

2. 用户的输入被传给模型中的函数。回调函数可以把用户输入的字符串转变成其他数据类型，例如符号、数等。

3. 模型函数返回的结果接着被放入消息框中，当然，先需要把它们转换成字符串。

程序的控制部分同时也负责窗口可见部分的绘制。在我们的教学软件包中，唯一有这个功能的函数是 create-window。标准的 GUI 工具箱提供了许多有此功能的函数，但是它们之间互不相同，而且它们自身往往也在迅速的改变中。

习题

习题 22.3.1　修改图 22.5 中的程序，使得它能够实现习题 5.1.2、习题 5.1.3 和习题 9.5.5 中的猜数字游戏。务必要做到玩家每次猜的数都由程序中的一个定义产生。

提示：可以使用习题 11.3.1 中能够产生随机数的函数。

习题 22.3.2　开发一个查找电话号码的程序。该程序的图形用户面应当包括一个文本框、一个消息框以及一个按钮。用户可以在文本框中输入人名，然后程序在消息框中给出电话号码。如果模型函数找不到这个人的电话号码（返回 false），则程序应在消息框中显示消息 name not found。

一般化你的程序，使得用户输入电话号码（也就是仅由数字组成的串），也能反过来查找人名。

提示：(1)Scheme 提供了函数 *string->symbol*，能把字符串转化为符号。(2)Scheme 还提供了函数 *string-> number*，能把字符串转化为数。如果该函数得到了一个不是数组成的字符串，它将返回 false：

```
  ( string-> number "6670004")
= 6670004

  ( string-> number "667-0004")
= false
```

这个一般化的过程示范了怎样将同样的 GUI 用于不同的模型。

真正的 GUI：图 22.2 中给出的图形用户界面并不是由我们的教学软件包建立的。实际上，教学软件包只提供了基本的 *gui-item*，不过，通过学习这些基本的 *gui-item*，完全可以掌握 GUI 程序设计的基本原理。设计真正的 GUI 涉及到图形设计以及产生 GUI 程序的工具（而不是手工设计 GUI）。

习题 22.3.3　开发函数 *pad->gui*，它读入一个标题（字符串）和一个 *gui-table*，然后把 *gui-table* 转化成 *gui-item* 表的表，即 *create-window* 能够接受的形式。*gui-table* 的定义如下：

> *cell* 是下列二者之一：
> 1. 数。
> 2. 符号。

> *gui-table* 是**(listof (listof** *cell***))**。

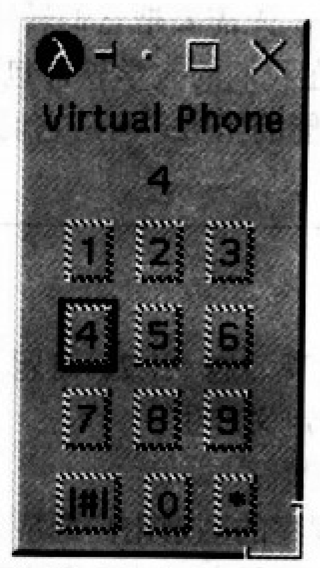
下面是 gui-table 的两个例子：

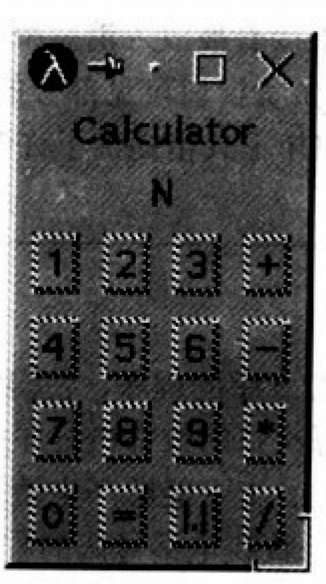
```
(define pad
  '((1 2 3)
    (4 5 6)
    (7 8 9)
    (\# 0 *)))
```

```
(define pad2
  '((1 2 3  +)
    (4 5 6  -)
    (7 8 9  *)
    (0 = \. /)))
```

左边的表格列出了一个电话键盘，右边的表格列出了一个计算器键盘。

函数 *pad->gui* 的返回值应当把每一个 cell 变成一个按钮，并且，该结果表的前两个元素应当是两条消息，第一条是标题，是不会改变的，第二条是用户最后按过的键。例如，上述两个 gui-table 的例子就应当产生如下的两个 GUI：





提示：其中的第二条消息需要一个初始值，例如"N"。

数学方面的例子



在实际的问题中，经常会使用到数学知识，这就要求在程序中实现数学函数。在许多情况下，这类程序会使用读入或者返回函数的函数。所以说，数学是一个很好的开端，通过它我们可以练习用函数编写程序，并练习建立抽象函数。

本章先讨论数学中一个非常重要的东西，即数列和级数；然后讨论积分，而积分主要是基于级数的；最后讨论函数的微分。

23.1 数列和级数

在代数学中，我们遇到过数列（有时数列也被称作级数）。下面就是三个数列的例子：

1. 0，2，4，6，8；
2. 1，3，5，7，9；
3. 5，10，15，20，25。

前两个例子分别是前五个偶数和前五个奇数；后一个例子是 5 的前五个倍数。数列也可以是无限的：

1. 0，2，4，6，8，……；
2. 1，3，5，7，9，……；
3. 5，10，15，20，25，……。

按照数学上的惯例，无穷数列以省略号结尾，读者必须自己判断该数列其余的项是什么。

一种理解数列，特别是无穷数列的方法是，把数列中的每个数按其序号和自然数对应起来。例如，奇数列和偶数列是这样与自然数对应的：

序号	0	1	2	3	4	5	6	7	8	9	……
偶数	0	2	4	6	8	10	12	14	16	18	……
奇数	1	3	5	7	9	11	13	15	17	19	……

从这张表中，我们很容易发现，第 i 个偶数就是 2 i，而第 i 个奇数就是 2 i+1。

所有这些表达都能很容易地翻译成简单的 Scheme 函数：

```
;; make-even : N -> N[even]
;; 计算第 i 个偶数
(define (make-even i)
  (* 2 i))
```

```
;; make-odd : N -> N[odd]
;; 计算第 i 个奇数
(define (make-odd i)
  (+ (* 2 i) 1))
```

简单说来，从自然数到数的函数就代表了一个无穷数列。

数学上，级数就是数列的总和。对我们前面举例所用的三个数列来说，它们的和分别是 20、25 和 75。对于无穷数列来说，考虑其有限部分（从第一个数开始，到某个数结束）的级数是很有趣的[1]。例如，自然数中，前 10 个偶数的和是 90，前 10 个奇数的和是 100。显然，计算级数是计算机做的工作。下面我们给出求前 n 个偶数的和以及求前 n 个奇数的总和的函数，这两个函数分别使用 *make-even* 和 *make-odd* 来计算所要求的偶数（或奇数）的值：

```
;; series-even : N -> number
;; 求前 n 个偶数的和
(define (series-even n)
  (cond
    [(= n 0) (make-even n)]
    [else (+ (make-even n)
             (series-even (- n 1)))]))
```

```
;; series-odd : N -> number
;; 求前 n 个奇数的和
(define (series-odd n)
  (cond
    [(= n 0) (make-odd n)]
    [else (+ (make-odd n)
             (series-odd (- n 1)))]))
```

自然，这两个函数需要抽象。下面是遵循基本抽象诀窍所得的结果：

```
;; series : N (N -> number) -> number
;; 求数列 a-term 前 n 项的和
(define (series n a-term)
  (cond
    [(= n 0) (a-term n)]
    [else (+ (a-term n)
             (series (- n 1) a-term))]))
```

在这个函数中，第一个参数确定从哪里开始做加法，第二个参数是从自然数到对应数列项的函数。为了测试 *series*，我们把它分别作用于 *make-even* 和 *make-odd*：

```
;; series-even1 : N -> number
(define (series-even1 n)
  (series n make-even))
```

```
;; series-odd1 : N -> number
(define (series-odd1 n)
  (series n make-odd))
```

一百多年来，数学家都使用希腊字母Σ表示级数。上述的两个级数可表示为：

$$\sum_{i=0}^{i=n} make-even(i) \qquad \sum_{i=0}^{i=n} make-odd(i)$$

一个真正的（或者懒惰的）数学家还会把 *make-even* 和 *make-odd* 替换成它们的定义，也就是 2 i 和 2 i+1。不过，为了强调良好的函数组织结构，我们不这么做。

---

[1] 某些无穷数列也有总和。具体说来，如果计算某个数列越来越多的项的总和，得到的数就越来越接近某个数，那么我们就称这个数为整个（无穷）数列的总和。例如，数列

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$$

的总和是 2。相反，数列

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$$

没有总和。

习题

习题 23.1.1 使用 **local** 语句，定义 *series-even* 和 *series-odd* 的抽象函数 *series-local*。用手工计算证明(*series-local make-even*)等价于 *series-even*。

23.2 等差数列和等差级数

等差数列

$$a_0, a_1, a_2, a_3, \ldots, a_n, a_{n+1}, \ldots$$

中，每一个项 a_{n+1} 都是前一个项 a_n 加上一个固定的常数的结果。下面就是一个（给出了自然数序号的）等差数列：

序　　号	0	1	2	3	……
等 差 数 列	3	8	13	18	……

在这个例子中，第一个数是 3，固定的常数是 5。前一个数又被称作初值，后一个数被称作公差。只要有了这两个数，我们就可以完全确定整个数列。

习题

习题 23.2.1 开发递归函数 *a-fives*，它读入一个自然数，用递归的方法，求出上例中等差数列的对应项。

习题 23.2.2 开发非递归函数 *a-fives-closed*，它读入一个自然数，求出上例中等差数列的对应项。有时非递归函数也被称作闭合函数。

习题 23.2.3 使用 *series*，求出 *a-fives* 数列的前 3 个、前 7 个以及前 88 个数的和。无穷等差数列的总和存在吗？

习题 23.2.4 开发函数 *seq-a-fives*，它读入自然数 *n*，返回数列 *a-fives* 或者 *a-fives-closed* 的前 *n* 项。提示：使用 build-list。

习题 23.2.5 开发 *arithmetic-series*。这个函数读入两个数，即 *start* 和 *s*，返回代表等差数列的函数，该等差数列的初值是 *start*，公差是 *s*。例如，(*arithmetic-series* 3 5)产生 *a-fives*（或者 *a-fives-closed*），(*arithmetic-series* 0 2)产生一个代表偶数数列的函数。

23.3 等比数列和等比级数

等比级数

$$g_0, g_1, g_2, g_3, \ldots, g_n, g_{n+1}, \ldots$$

中，每个项 g_{n+1} 都是前一个项 g_n 乘上一个固定的常数的结果。下面就是一个（给出了自然数序号的）等比数列：

序　　号	0	1	2	3	4	……
等 差 数 列	3	15	75	375	1875	……

在这个例子中，第一个数是 3，固定的常数是 5。前一个数又被称作初值，后一个数被称作公比。只要有了这两个数，整个数列就可以完全被确定。

习题

习题 23.3.1 开发递归函数 *g-fives*，它读入一个自然数，用递归的方法，求出上例中等比数列的对应项。

习题 23.3.2 开发非递归函数 *g-fives-closed*，它读入一个自然数，求出上例中等比数列的对应项。

习题 23.3.3 开发函数 *seq-g-fives*，它读入自然数 *n*，求出数列 *g-fives* 或者 *g-fives-closed* 的前 *n* 项。提示：使用 build-list。

习题 23.3.4 开发 *geometric-series*。这个函数读入两个数，即 *start* 和 *s*，返回代表等比数列的函数，该等比数列的初值是 *start*，公比是 *s*。例如，(*geometric-series* 3 5))产生 *g-fives*（或者 *g-fives-closed*）。

习题 23.3.5 使用 *series*，求出 *g-fives* 数列的前 3 个、前 7 个以及前 88 个数的和。使用 *series*，给出(*geometric-series* 1 .1)的前 3 个、前 7 个以及前 88 个数的和。无穷等比数列的总和存在吗？

泰勒级数

像 π 和 *e* 这样的数学常数以及像 sin、cos 和 log 这样的数学函数都很难计算。不过，这些函数在许多的工程应用中十分重要，所以数学家化了很多时间和精力寻找好的方法来计算这些函数的值。有一种方法，是用这些函数的泰勒级数代替它们，而泰勒级数简单来说就是一种无穷多项式。

泰勒级数就是一个序列的和。与等差数列和等比数列不同，泰勒级数的每一个项取决于两个未知量：变量 x 以及其在序列中的位置 i。指数函数的泰勒级数是：

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

换句话说，如果我们想要计算 e^x（其中的 x 是任意一个我们已经知道的值），我们可以把上述泰勒级数中的 x 替换成我们已知的那个值，然后计算级数的值。例如，假设 x 是 1，于是我们把泰勒级数中的 x 替换成 1，泰勒级数就变成了一个普通级数（即，由数组成的级数）：

$$e^1 = 1 + \frac{1}{1!} + \frac{1^2}{2!} + \frac{1^3}{3!} + \ldots$$

这个级数是一个无穷数列的总和，实际上，它也是一个数，并且，只要把数列中的前几项加起来，我们就能大概知道这个数是多少。

计算泰勒级数的关键步骤是用 x 和位置 i 的函数表达级数的每个项。在我们的例子中，指数函数的泰勒级数的每个项的形式是：

$$\frac{x^i}{i!}.$$

假设 x 是固定的，下面给出了一个等价的 Scheme 定义：

```
;; e-taylor : N -> number
(define (e-taylor i)
  (/ (expt x i) (! i)))
;; ! : N -> number
(define (! n)
  (cond
```

```
[(= n 0) 1]
[else (* n (! (sub1 n)))])))
```

前一个函数计算泰勒级数每个项的值；后一个函数计算自然数的阶乘。要计算 e^x 的值，只需求出(*series* 10 *e-taylor*)的值（假设我们只计算泰勒级数的前 10 项）。

把所有的东西都集中起来，就可以定义一个计算 *e* 的 *x* 次方的函数。既然这个函数需要两个辅助函数，我们使用 local：

```
(define (e-power x)
  (local ((define (e-taylor i)
        (/ (expt x i) (! i)))
      (define (! n)
        (cond
          [(= n 0) 1]
          [else (* n (! (sub1 n)))])))
    (series 10 e-taylor)))
```

习题

习题 22.3.6　在 *e-power* 的定义中，把 10 替换成 3，然后手工计算(*e-power* 1)。只显示那些包含 *e-laylor* 对数的新调用的行。

e-power 的返回值是个分数，而且，该分数的分母往往很大。与之不同，Scheme 内建的 exp 函数生成不精确的数值。当然，我们可以使用如下的函数把精确的数值转化为不精确的数值：

```
;; exact-> inexact : number [exact] -> number [inexact]
```

测试这个函数，并把它加入到 *e-power* 的主体之中，然后比较 exp 和 *e-power* 的返回值。增加级数的项，直到两个函数的返回值之差变得非常小。

习题 23.3.7　开发函数 *ln*，计算自然对数的泰勒级数。自然对数的泰勒级数是：

$$ln(x) = 2 \cdot \left[\left(\frac{x-1}{x+1} \right) + \frac{1}{3} \cdot \left(\frac{x-1}{x+1} \right)^3 + \frac{1}{5} \cdot \left(\frac{x-1}{x+1} \right)^5 \cdots \right]$$

对于任何一个大于 0 的 *x*，该泰勒级数都有一个确定的值。

DrScheme 也提供了 log，即计算自然对数的基本操作。比较 ln 和 log 的返回值，然后使用 exact-> inexact（参见习题 23.3.6）使结果更为易于比较。

习题 23.3.8　开发函数 *my-sin*，计算三角函数 sin 的泰勒级数。sin 的泰勒级数是：

$$sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots$$

对于任意的 *x*，该泰勒级数都有意义。

提示：在 sin 的泰勒级数中，奇数项是正的，偶数项是负的。数学家通过计算 $(-1)^i$ 来确定符号；程序员则可以使用 cond 来决定符号。

习题 23.3.9　数百年来，数学家们都是使用级数来计算 π 的。最早的一个级数是由格雷戈里（1638—1675）发现的：

$$\pi = 4 \cdot \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots \right].$$

定义函数 *greg*，把一个自然数映射成该级数对应的项，然后使用 *series* 给出 π 的近似值。

注意：随着级数的项不断增加，该近似值逐渐接近 π。不幸的是，用这种方法计算 π 实际上是行

不通的。

23.4 函数曲线下方的面积

考虑图 23.1 中的函数图形，假设我们想知道位于 *x* 轴、粗线 *a*、粗线 *b* 和函数曲线之间的图形的面积。测定在某个区间中函数下方的面积被称作对该函数积分。在计算机出现之前，工程师们就必须求解这类问题，所以数学家们详尽地研究过它。对于一小部分函数，我们可以精确地给出这个区域的面积，而对于其他的函数，数学家们发展了一种方法，可以给出非常接近的近似值。这种方法需要进行大量的机械性运算，所以它们很适合用计算机来计算。

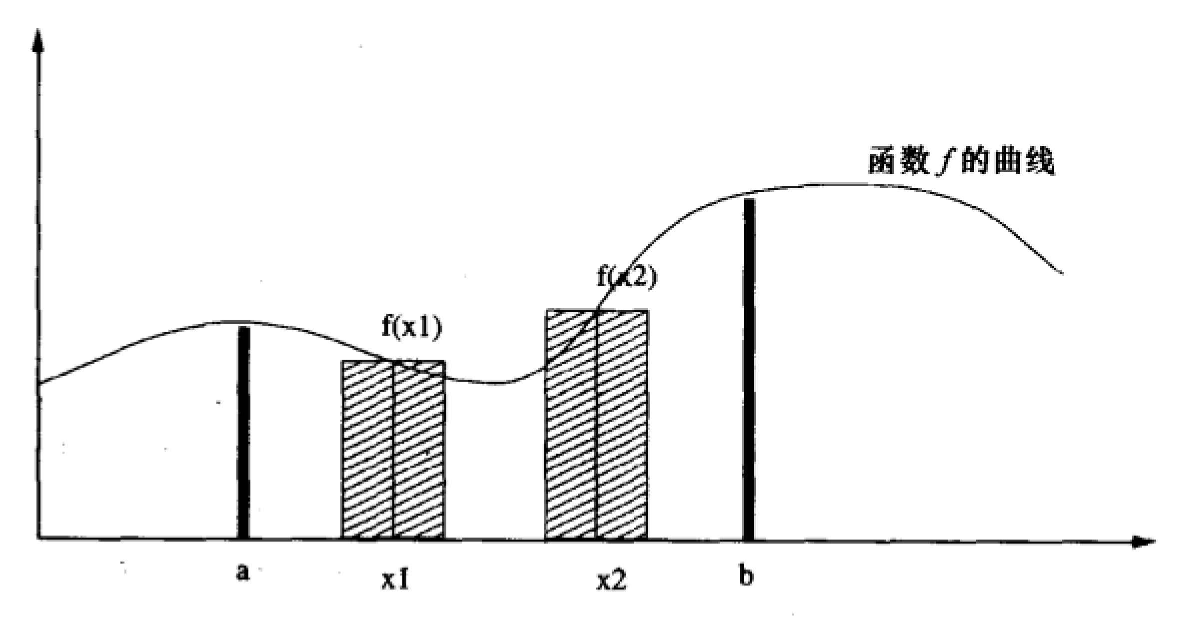


图 23.1 在 a 和 b 之间对一个函数积分

一般来说，积分函数需要三个参数：*a*、*b* 以及函数 *f*。区域的第四条边界——x 轴——总是被省略掉。于是，我们可以得出如下的合约：

```
;; integrate : (number -> number) number number -> number
;; 计算在 a 和 b 之间 f 的图形之下的面积
(define (integrate f a b) ...)
```

开普勒提出了一种简单的积分方法，由下列三步组成：

1. 区间分成两部分：[*a*，（*a*+*b*）/2]以及[（*a*+*b*）/2，*b*]；
2. 计算这两个梯形的面积；
3. 把两个部分的面积加起来，得到一个积分的估计值。

习题

习题 23.4.1 开发函数 *integrate-kepler*，使用开普勒的方法计算函数 *f* 在 *left* 和 *right* 之间的积分。

另一个简单的方法是把该区域看作是由许多窄矩形组成的，每一个小矩形的高就是在矩形中部的函数值。图 23.1 中显示了两个这样的矩形。把所有的矩形的面积加在一起，就得到了函数曲线下方面积的一个估计值。小矩形的数量越多，这个估计值就越接近真正的积分值。

假设 *R* 代表矩形的数量，为了求出每个矩形的面积，我们还需要知道它们的边长。矩形在 *x* 轴上的边长是整个区间的长度除以矩形的数量：

$$width = \frac{(b-a)}{R}.$$

要想知道矩形的高，我们还需要给出每个矩形底边的中点以及在中点处 f 的值。显然，第一个矩形的中点是 a 加上一半的底边长。所以，如果

$$step = \frac{width}{2},$$

那么，第一个矩形的面积就是：

$$w \cdot f(a+s),$$

其中 W 代表矩形的宽度，S 代表步长。

从第一个矩形（从 a 开始的那个）向右移动一格，就是第二个矩形。所以说，第二个矩形的中点是在第一个矩形的中点上加上一倍的宽度。换句话说，下一个中点（在图 23.1 中，就是点 x1）位于

$$a + w + s,$$

第三个中点位于

$$a + 2 \cdot w + s,$$

以此类推。下面的表格给出了前三个矩形的参数：

序号	0	1	2	……
中点	$a+S$	$a+1 \cdot W+S$	$a+2 \cdot W+S$	……
在中点处的 f 值	$f(a+S)$	$F(a+1 \cdot W+S)$	$f(a+2 \cdot W+S)$	……
面积	$W \cdot f(a+S)$	$W \cdot f(a+1 \cdot W+S)$	$W \cdot f(a+2 \cdot W+S)$	……

第一个矩形的序号是 0，最后一个矩形的序号是 $R-1$。

使用这一系列的矩形，现在我们可以使用如下的级数来求出区域的面积：

$$\sum_{i=0}^{i=R-1} area-of-yec\tan gle(i) \cdot$$

习题

习题 23.4.2　开发函数 *integrate*，使用矩形级数的方法计算在 *left* 和 *right* 之间函数 f 下方的面积。

测试 *integrate*。测试中使用的 f、a 和 b 应当能够通过简单的手工计算给出积分的值，例如，(define (id *x*) *x*)。比较 *integrate* 的结果和习题 23.4.1 的结果。

把 *R* 定义成最外层常数：

;; *R*：用来近似积分的矩形的数目

(define *R* ...)

使用 sin 测试 *integrate*，并且把 *R* 逐步地从 10 增加到 10000。结果有何变化？

23.5　函数的斜率

我们换一种角度来观察图 23.1 中的函数图形。在许多问题中，我们所需要的是求出函数在某一点处的切线——也就是通过该点，并且在该点与函数有着相同斜率的直线。这时，问题的本质是要计算出函

数的斜率。在经济学方面，如果函数曲线代表一个公司在不同时期的收入，那么斜率就代表该公司的发展速度；在物理学方面，如果函数曲线代表某个物体的速度，那么斜率就代表该物体的加速度。

求函数 f 在某一点的斜率被称为函数的*微分*。微分算子（算子又称泛函）返回一个函数 f'（称为 f 的导函数），在每一个点 x，f'的值就是 f 在该点的斜率。计算 f'的过程很复杂，所以这又是一个适合计算机程序做的工作。这个程序的输入是 f，输出是 f'。

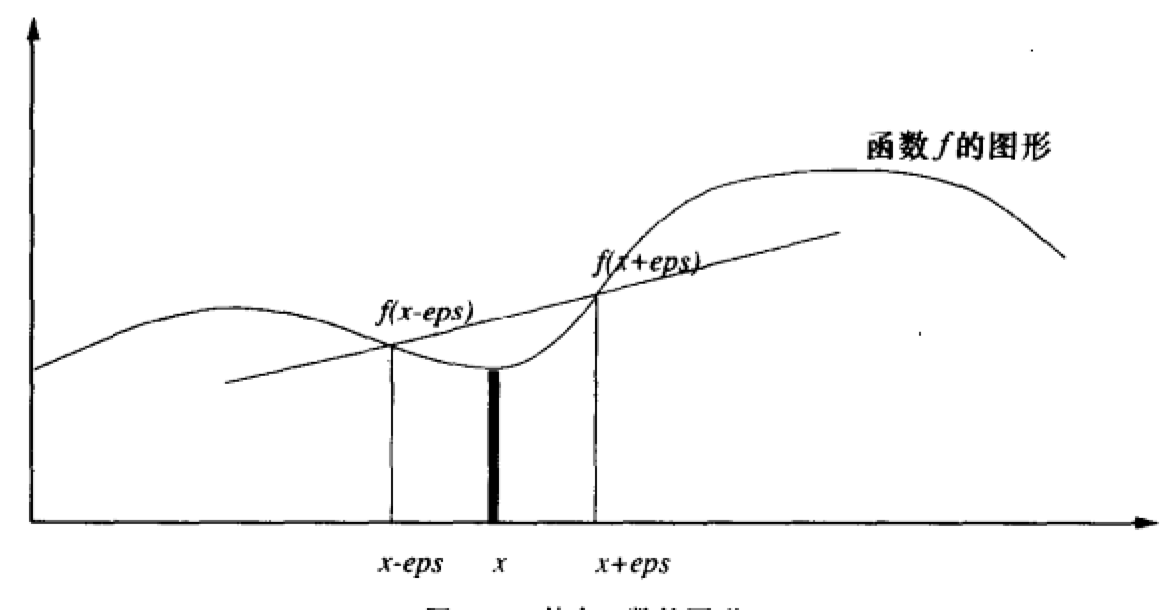


图 23.2　某个函数的图形

要设计“微分器”，我们必须先研究如何能找到（在某一个点）与函数曲线有着相同斜率的直线。原则上，这样的直线(在该点附近)与函数曲线只有一个交点，但是我们暂时放松这个条件，考虑在该点附近与函数曲线有两个交点的直线。我们把这两个交点设在与 x 等距的地方，也就是 $x-\varepsilon$ 和 $x+\varepsilon$ 处，这里的常数 ε 代表一个非常小的距离。有了这两个点，我们就能完全确定一条直线，并且得到该直线的斜率。

图 23.2 是描述这种情形的草图。如果我们要求函数 f 在 x 点的微分，相邻的两点就是（$x-\varepsilon$，f（$x-\varepsilon$））和（$x+\varepsilon$，f（$x+\varepsilon$）），于是所确定的直线的斜率是：

$$\frac{f(x+e)-f(x-e)}{2\cdot e}.$$

更确切地说，就是这两点的高度差除以它们之间的水平距离。接下来的问题留作习题，分别是通过一个已知点和斜率求出一条直线，以及通过两个已知点求出直线。

习题

习题 23.5.1　直线的方程是：

$$y(x)=a\cdot x+b.$$

现在，我们能够直接把该等式翻译成 Scheme 语句：

```
(define (y x)
  (+ (* a x) b))
```

要得到一条具体的直线，我们必须把 a 和 b 替换成数。

教学软件包 graphing.ss 提供了一个绘制直线的操作 *graph-line*。该操作的参数包括直线 y 及其颜色，例如'red。使用 *graph-line* 绘制下列直线：

1. $y_1(x) = x + 4$
2. $y_2(x) = 4 - x$
3. $y_3(x) = x + 10$

4. $y_4(x)$ = 10 - x
5. $y_5(x)$ = 12

习题 23.5.2 给定一个点和斜率，建立直线方程是一道标准的数学习题。到你的数学教科书中找出求解的方法，然后开发函数 *line-from-point+slope*，实现该方法。这个函数读入 *posn*（点）和数（斜率），返回表示直线的函数，其形式如同习题 23.5.1 中给出的表示直线的函数。

有两种方法可以测试函数生成的函数。就 *line-from-point+slope* 的情形而言，假设我们给它的输入是（0，4）和 1，其结果应当是习题 23.5.1 中的直线 y_1。要检查这一点，我们可以把

```
(line-from-point+slope (make-posn 0 4) 1)
```

返回的结果作用到一些数上，也可以使用 graphing.ss 提供的操作绘制该直线。如果使用第一种方法，我们必须把得到的结果与 y_1 比较；如果使用第二种方法，我们可以使用某种颜色绘制 *line-from-point+slope* 返回的直线，用另一种颜色绘制手工建立的 y_1，再观察结果。

一旦得到了过（$x-\varepsilon$，$f(x-\varepsilon)$）和（$x+\varepsilon$，$f(x+\varepsilon)$）两点的直线，我们就能求出该直线的斜率。逐步减小 ε 的值，使它不断地接近 0，则（$x-\varepsilon$，$f(x-\varepsilon)$）和（$x+\varepsilon$，$f(x+\varepsilon)$）两个点也逐步靠近。直到最后，它们变成了同一个点，即（x，$f(x)$），我们想求斜率的那个点[1]。

习题

习题 23.5.3 使用教学软件包 graphing.ss 提供的操作 *graph-fun* 绘制数学函数

$$y(x)=x^2-4\cdot x+7.$$

该操作的使用方法与 *draw-line*（参见习题 23.5.1）完全相同。

假设我们想要知道该函数在点 $x=2$ 的斜率。挑选一个大于 0 的 ε，然后求出经过（$x-\varepsilon$，$f(x-\varepsilon)$）和（$x+\varepsilon$，$f(x+\varepsilon)$）两点的直线的斜率（其中的 f 就是上述公式中的 $y(x)$）。使用习题 23.5.2 中的 *line-from-point+slope* 求出直线方程，然后使用 *draw-line* 在同一个坐标系中绘制出该直线。最后，使用 ε/2 以及 ε/4 重复该过程。

如果我们的目的是定义一个 Scheme 函数，实现微分算子，那么，我们可以把 ε 设为一个较小的数值，然后把上述数学公式翻译成 Scheme 表达式：

```
;; d/dx : (num -> num) -> (num -> num)
;; 使用数值方法，计算 f 的导函数
(define (d/dx f)
  (local ((define (fprime x)
            (/ (- (f (+ x ε)) (f (- x ε)))
               (* 2 ε)))
          (define ε ...))
    fprime))
```

注意 *d/dx* 的输入是一个函数，输出也是一个函数——就和数学中的微分算子一样。

正如我们在本节一开始所提到的，微分算子从某个函数 f 计算出其导函数 f'，而对于任意一个 x，$f'(x)$ 给出函数 f 在 x 点的斜率。对于直线来说，其斜率是已知的，所以用直线来测试 *d/dx* 再好不过了。考虑：

[1] 逐步减小 ε 的值的过程是一个收敛（或者极限）过程。该过程并不保证成功，也就是说，对于某些函数这个过程不收敛。这里我们忽略这一点，但是这是一种常见的现象，需要给予特别的注意。

```
(define (a-line x)
  (+ (* 3 x) 1))
 (d/dx a-line)的计算过程是：
 (d/dx a-line)
= (local ((define (fprime x)
            (/ (- (a-line (+ x ε)) (a-line (- x ε)))
               (* 2 ε)))
          (define ε ...))
    fprime)

= (define (fprime x)
    (/ (- (a-line (+ x ε)) (a-line (- x ε)))
       (* 2 ε)))
  (define ε ...)
  fprime
```

现在，如果把(+ x ε)和(- x ε)当作数，我们就可以在 *fprime* 的定义中计算 *a-line* 的调用了[1]：

```
 (define (fprime x)
   (/ (- (+ (* 3 (+ x ε)) 1) (+ (* 3 (- x ε)) 1))
      (* 2 ε)))
= (define (fprime x)
   (/ (- (* 3 (+ x ε)) (* 3 (- x ε)))
      (* 2 ε)))
= (define (fprime x)
   (/ (* 3 (- (+ x ε) (- x ε)))
      (* 2 ε)))
= (define (fprime x)
   (/ (* 3 (* 2 ε))
      (* 2 ε)))
= (define (fprime x)
   3)
```

换句话说，(*d/dx* *a-line*)返回的结果总是 3，即 *a-line* 的斜率。简而言之，通过使用较小的 ε，我们不只是得到了微分的近似值，而实际上得到了正确的答案。不过，一般而言，这样得到的结果取决于 ε，并且是不精确的。

习题

习题 23.5.4 选用一个较小的 ε，用 *d/dx* 计算下面的函数在点 $x=2$ 的斜率：

$$y(x)=x^2-4\cdot x+7$$

比较这个结果与习题 23.5.3 的计算，有何异同？

习题 23.5.5 开发函数 *line-from-two-points*，它读入点 *p1* 和 *p2*，返回的 Scheme 函数代表经过 *p1* 和 *p2* 的直线。

[1] 如果 x 是数的话，那么 x+ε 和 x−ε 也都是数。但是，如果我们不小心把 *fprime* 作用于别的东西，那么这两个表达式（指 x+ε 和 x−ε）都会产生错误信号。

问题：是否有该函数不能计算出函数的情形。如果有的话，修改定义以便在出现这种情形时产生一个适当的错误消息。

习题 23.5.6　计算如下函数在点 $x=4$ 的斜率：

```
(define (f x)
  (+ (* 1/60 (* x x x))
     (* -1/10 (* x x))
     5))
```

分别把 ε 的值设为 2、1 和.5。最后，对于一些其他的 x 值，计算微分（同样把 ε 的值分别设为 2、1 和 5）。



定义匿名函数



使用抽象函数时，在许多情况下都需要定义辅助函数。例如 *filter1*，它读入一个过滤函数、一个表以及一个过滤元素。就在前一章中，有三次使用 *filter1* 时也分别使用了辅助函数：*squared?*、$<_{ir}$和 *eq-ir?*。

既然这些辅助函数仅仅被用作 *filter1* 的参数，我们应当使用第 18 章中给出的程序组织原则。也就是说，应当使用一个 local 表达式，把对 *filter1* 的调用和辅助函数的定义结合在一起。下面就是 *filter1-eq-ir?* 的一种定义：

```
;; find : list-of-IRs symbol -> boolean
(define (find aloir t)
  (local ((define (eq-ir? ir p)
           (symbol=? (ir-name ir) p)))
    (filter1 eq-ir? aloir t)))
```

另一种可行的定义是把 local 表达式放在需要使用该函数的地方：

```
;; find : list-of-IRs symbol -> boolean
(define (find aloir t)
  (filter1 (local ((define (eq-ir? ir p)
             (symbol=? (ir-name ir) p)))
           eq-ir?)
              aloir t))
```

这种定义更为合理，因为函数的名字（例如 *eq-ir?*）现在是合法的表达式，因而能够作为 local 的主体。这样，local 表达式就引入了一个函数定义，并以该函数为返回值。

优秀的程序设计者都使用抽象函数，并且用一种简洁的方式来组织程序，所以，Scheme 对这种特别的、常用的 local 提供了专门的简写。这种简写被称作 lambda 表达式，它极大地方便了类似于 *eq-ir?*、*squared?*或$<_{ir}$这类函数的引入。本章的前两节分别介绍 lambda 表达式的语法和语义，最后一节讨论其语用。

24.1 lambda 表达式的语法

lambda 表达式是一种新的表达式：

```
<exp> = (lambda (<var> ... <var>) <exp>)
```

它与其他表达式的区别之处就是关键字 lambda，lambda 的后面跟着由括号包含的变量序列，最后一部分则是一个表达式。

下面是 lambda 表达式的三个例子：

1. `(lambda(xc)(>(*xx)c))`
2. `(lambda(irp)(<ir-price ir)p))`

```
3. (lambda(ir p)(symbol=?(ir-name ir)p))
```

它们分别对应于前面提到的 *squared?*、$<_{ir}$和 *eq-ir?*。

lambda 表达式定义了一个匿名函数，也就是一个没有名字的函数。关键字 lambda 后面跟着的变量序列就是这个函数的参数，最后的表达式部分就是函数的主体。

24.2 lambda 表达式的辖域和语义

正如本章的引言所介绍的，**lambda** 表达式就是 **local** 表达式的简写。一般来说，我们把

```
(lambda (x-1 ... x-n) exp)
```

理解为

```
(local ((define (a-new-name x-1 ... x-n) exp))
  a-new-name)
```

其中，函数的名字 *a-new-name*（一个新的名字）不可以在 exp 中出现。

简写的解释表明，

```
(lambda (x-1 ... x-n) exp)
```

引入绑定变量 *x-1...x-n*，它们的辖域就是参数 exp。当然，如果 exp 中含有更多的绑定结构（例如，一个嵌套的 local 表达式），那么这些变量的辖域可能会有缺口。

这个解释还间接地表达了 lambda 表达式求值的基本要素：

1. 因为函数是值，所以 lambda 表达式也是值。

2. 把 lambda 表达式作用于某些值，就好像是把一个函数作用于参数。如果先把简写展开，计算过程完全符合普通的函数调用的规则。

下面就是一个使用 **lambda** 的简单例子：

```
(filter1 (lambda (ir p) (< (ir-price ir) p))
         (list (make-ir 'doll 10))
          8)
```

这里，*filter1* 的调用使用 lambda 表达式、一个（很短的）存货记录表以及一个限值作为参数。我们可以把 lambda 表达式转换成 local 表达式，用来理解计算的过程：

```
...
= (filter1 (local ((define (<ir ir p) (< (ir-price ir) p))) <ir)
           (list (make-ir 'doll 10))
            8)
= (filter1 <ir
           (list (make-ir 'doll 10))
            8)
```

在这步计算中，$<_{ir}$被提取出 local 表达式，放入最外层。接下来的计算过程与第 19.2 节给出的一样。

虽然把 lambda 表达式理解成简写很自然，但是上述的计算暗示我们可以直接理解 lambda 表达式。特别地，我们可以这样修改 lambda 表达式的计算规则：

((**lambda** (`x-1...x-n`) **exp**) val `val-1...val-n`) = **exp** 其中所有的 `x-1 ... x-n` 都被替换成 `val-1 ... val-n`

即，lambda 表达式的调用就跟普通表达式一样，把函数的所有参数都替换成实际的值，然后计算函

数的主体。

用这种方法处理上面的例子：

```
(filter1 (lambda (ir p) (< (ir-price ir) p))
         (list (make-ir 'doll 10))
          8)
```

一般来说，该调用会先使用 *filter1* 的主体替换整个表达式，同时把 *filter1* 的所有参数都替换成实际的值。这个步骤把整个 lambda 表达式当作 *filter1* 的一个参数：

```
= (cond
    [((lambda (ir p) (< (ir-price ir) p))
      (make-ir 'doll 10) 8)
     (cons (first (list (make-ir 'doll 10)))
       (filter1 (lambda (ir p) (< (ir-price ir) p))
                (rest (list (make-ir 'doll 10)))
                8))]
    [else
      (filter1 (lambda (ir p) (< (ir-price ir) p))
               (rest (list (make-ir 'doll 10)))
               8)])
= (cond
    [(< (ir-price (make-ir 'doll 10)) 8)
     (cons (first (list (make-ir 'doll 10)))
       (filter1 (lambda (ir p) (< (ir-price ir) p))
                (rest (list (make-ir 'doll 10)))
                8))]
    [else
      (filter1 (lambda (ir p) (< (ir-price ir) p))
               (rest (list (make-ir 'doll 10)))
               8)])
= ...
```

接下来的计算过程就跟以前一样了。不过，这个简短的计算过程还表明，虽然在程序中使用 **lambda** 表达式非常方便，但用一个有名字的函数代替它总能简化计算过程。

习题

习题 24.2.1 判断下列哪些语句是合法的 lambda 表达式：

1. `(lambda (x y) (x y y))`
2. `(lambda () 10)`
3. `(lambda (x) x)`
4. `(lambda (x y) (x))`
5. `(lambda x 10)`

解释它们为什么是合法的，或者为什么不合法。

习题 24.2.2 在下列三个 lambda 表达式中，用箭头把每个带下划线的 *x* 连接到与之对应的绑定变量：

1.

```
(lambda (x y)
   (+ x (* x y)))
```

2.

```
(lambda (x y)
   (+ x
    (local ((define x (* y y)))
      (+ (* 3 x)
         (/ 1 x)))))
```

3.

```
(lambda (x y)
   (+ x
     ((lambda (x)
        (+ (* 3 x)
           (/ 1 x)))
       (* y y))))
```

另外，对于每一个带下划线的 x，划出其辖域（注意辖域内可能包含缺口）。

习题 24.2.3　手工计算下列表达式：

1.

```
((lambda (x y)
   (+ x (* x y)))
1 2)
```

2.

```
((lambda (x y)
   (+ x
       (local ((define x (* y y)))
         (+ (* 3 x)
            (/ 1 x)))))
   1 2)
```

3.

```
((lambda (x y)
    (+ x
     ((lambda (x)
        (+ (* 3 x)
           (/ 1 x)))
      (* y y))))
   1 2)
```

24.3 lambda 表达式的语用

使用 lambda 表达式的原则相当简明易懂：

> **使用 lambda 表达式的原则**
>
> 如果某个非递归函数只需要当参数使用一次，那么使用 lambda 表达式。

假如前面章节中的所有程序都按照这个原则来组织，那么我们就会发现 lambda 极大地简化了抽象函数的使用。正因为如此，Scheme 在其库中提供了大量的抽象函数。在以后的章节中，我们会遇到更多体现 lambda 表达式简便之处的例子。



第五部分

生成递归





一种新的递归形式



到目前为止，我们所开发的函数可以分为两种主要类型，一种是封装某个领域的知识的函数，另一种是处理结构体或表的函数。这第二种函数的特点是，它们一般把参数分解成直接的结构组成成分，然后处理这些成分。如果某个成分与输入（参数）属于同一数据类型，那么这个函数就是一个递归函数。因为这个原因，我们把此类函数称为（结构）递归函数。不过在某些情况下，我们还需要基于另一种形式的递归，它被称为生成递归。这种形式的递归与数学一样古老，也被称为算法。

算法的输入代表一个问题。通常，该问题是一大类问题中的一个实例，而算法能够处理所有这些问题。一般来说，算法把问题分割成更小的问题，然后解决这些小问题。例如，计划一个假期旅行的算法需要安排从家到最近的机场的行程、从该机场到与度假村最近的机场的飞行路线以及从机场到度假村旅馆的行程。整个问题的解就是通过把这些小问题的解结合起来得到的。

设计一个算法需要区分两种类型的问题：平凡可解的问题[1]和非平凡可解的问题。如果某个给定的问题是平凡可解的，算法就可以给出相应的解。例如，从我们的家到最近的机场的问题是平凡可解的。我们可以开车去，乘出租车去，或者请朋友驾车送我们去。如果某个给定的问题是非平凡可解的，算法会产生新的问题，然后解决这些新问题。长途旅行就是一个非平凡（可解问题）的例子，它可以通过产生新的、更小的问题来得到解决。从计算的角度说，每个新的小问题往往与原来的问题属于同一类型，正是因为这个原因，我们把这种解决问题的步骤称为生成递归。

在本书的这一部分，我们学习设计算法，即基于生成递归的函数。从这种思想的描述中，我们知道，与结构递归函数的数据驱动设计相比，算法的设计是相当特殊的行为。事实上，我们最好称之为发明一种算法，而不是设计一种算法。发明算法需要一种新的洞察力。有时候，这个过程几乎不需要洞察力。例如，求解一个“问题”可能只需要列举一系列数。不过，也有时候，它可能会依赖于某个数学定理。要完全理解这种设计过程，我们有必要先来学习一些例子，从而理解不同类型的问题。实际上，新的、复杂的算法通常总是由数学家和理论计算机科学家设计的；不过，程序设计者必须大致了解基本的思想，这样他们才能自己创造简单的算法，并从科学家那里学习复杂的算法。

这一章列举了两个完全不同的算法：前者是程序设计者在日常工作中创造的算法；后者是一种快速排序算法——生成递归在计算领域的一个早期应用。

术语：理论计算机科学家通常不区分结构递归和生成递归，而是把这两种函数都称为算法。有时，他们使用术语“递归”和“迭代”，其中后者表示这样一类函数定义，其中递归函数的调用位于定义中某个特定的位置。在本书中，我们会严格地使用“算法”和“生成递归”这两个术语，因为，比起应用数学家纯粹按照句法来分类，这种分类方法更适合设计诀窍的思想。

[1] 在本书的这一部分中，“平凡”是一个专用术语。第 26 章会给出它的解释。

25.1 为桌上的一个球建立模型

我们来考虑一个看上去很简单的问题：为一个在桌面上移动的小球建立模型。假设这个小球以恒定的速度在桌面上移动，直至它掉出桌面为止。我们可以把桌面模型化为一个有着固定宽度和高度的画布。小球就是在画布上移动的一个圆盘，我们通过绘制该圆盘、等待、再清除该圆盘来表示小球，直到它离开限定的区域为止。

图 25.1 种给出了建立模型所需的函数、结构体、数据和变量：

```
;; 教学软件包: draw.ss

(define-struct ball (x y delta-x delta-y))
;; ball 是结构体:
;;    (make-ball number number number number)

;; draw-and-clear : a-ball -> true
;; （在画布上）绘制、休眠、清除一个圆盘
;; 结构的设计，Scheme 知识
(define (draw-and-clear a-ball)
  (and
   (draw-solid-disk (make-posn (ball-x a-ball) (ball-y a-ball)) 5 'red)
   (sleep-for-a-while DELAY)
   (clear-solid-disk (make-posn (ball-x a-ball) (ball-y a-ball)) 5 'red)))

;; move-ball : ball -> ball
;; 建立一个新的小球，对 a-ball 的一步移动建模
;; 结构的设计，物理知识
(define (move-ball a-ball)
  (make-ball (+ (ball-x a-ball) (ball-delta-x a-ball))
             (+ (ball-y a-ball) (ball-delta-y a-ball))
             (ball-delta-x a-ball)
             (ball-delta-y a-ball)))

;; 画布的尺寸
(define WIDTH 100)
(define HEIGHT 100)
(define DELAY .1)
```

图 25.1 *move-until-out* 的辅助函数

1．球是结构体，它包含四个字段：两个方向上的当前位置和速率。也就是说，*ball* 结构体中的前两个数是球在画布上的位置，后两个数分别描述球在两个方向上每一步移动的量；

2．函数 *move-ball* 模拟球的物理运动，读入一个球，建立一个新的、移动了一步的球；

3．函数 *draw-and-clear* 先在当前位置绘制小球，然后等待一小段时间，最后把小球清除。

变量定义指定了画布的尺寸以及延迟的时间。

要把球移动几小步，可以这样写：

```
(define the-ball (make-ball 10 20 -5 +17))
(and
 (draw-and-clear the-ball)
 (and
```

```
  (draw-and-clear (move-ball the-ball))
  ...))
```

尽管这样写很单调乏味。取而代之的是，应当开发一个函数，移动小球直到它完全离开画布的范围。比较简单的工作是先定义函数 *out-of-bounds?*。这个函数测定给定的小球在画布上是否还可见：

```
;; out-of-bounds? : a-ball -> boolean
;; 测定 a-ball 是否超出边界
;; 领域知识，几何
(define (out-of-bounds? a-ball)
  (not
   (and
    (<= 0 (ball-x a-ball) WIDTH)
    (<= 0 (ball-y a-ball) HEIGHT))))
```

在前面的章节中，我们已经定义了大量类似于 *out-of-bounds?*的函数了。

相反，编写一个函数，在画布上绘制小球，直到它出界为止，这属于我们目前还没有遇到过的一类问题。我们从这个函数的基础部分开始：

```
;; move-until-out : a-ball -> true
;; 对小球的移动建模，直到它出界为止
(define (move-until-out a-ball) ...)
```

因为该函数读入一个球并在画布上绘制其移动，它应该像其他的画图函数一样返回 true。然而，使用针对结构体的设计诀窍来设计它是毫无意义的。毕竟，现在我们已经明白怎样使用 *draw-and-clear* 来绘制并清除小球，也清楚怎样移动它。所以我们需要的是对情况进行区分，检查小球是否还在界内。

我们改进函数的头部，添上一个合理的 cond 表达式：

```
(define (move-until-out a-ball)
  (cond
    [(out-of-bounds? a-ball) ...]
    [else ...]))
```

我们已经定义过 *out-of-bounds?*函数了，因为在问题的描述中，“离开指定的范围”是一个单独的概念。

如果 *move-until-out* 读入的球不在画布的范围之内，按照合约函数可以返回 true。如果小球还在界内，必然会发生两件事：一，必须在画布上画出小球，再把它清除；二，小球必然会移动，然后再做同样的事。这意味着，在移动小球后，我们要再次调用 *move-until-out*，也就是说该函数是递归的：

```
;; move-until-out : a-ball -> true
;; 对小球的移动建模，直到它出界为止
(define (move-until-out a-ball)
  (cond
    [(out-of-bounds? a-ball) true]
    [else (and (draw-and-clear a-ball)
               (move-until-out (move-ball a-ball)))]))
```

*(draw-and-clear a-ball)*和*(move-until-out (move-ball a-ball))*都返回 true，而且这两个表达式都需要被求值，所以我们用 and 表达式连接它们。

现在可以测试这个函数：

```
(start WIDTH HEIGHT)
(move-until-out (make-ball 10 20 -5 +17))
```

```
(stop)
```

它会创建一个大小合适的画布，以及一个向左下方移动的球。

仔细研究这个函数，会发现它有两个特点。第一，虽然这个函数是递归的，但是函数的主体是由一个 cond 表达式组成的，其中的条件与输入数据无关；第二，在函数主体中，递归调用并不使用输入的某个部分作参数，而是使用 *move-until-out* 生成一个全新的、与原来不同的 *ball* 结构体，代表原来的球在一步移动后的结果，然后再递归。显然，没有哪个设计诀窍可以产生这样一个定义，我们遇到了一种新的编程方法。

习题

习题 25.1.1 如果把下面这三个表达式：

```
(start WIDTH HEIGHT)
(move-until-out (make-ball 10 20 0 0))
(stop)
```

放在 Definitions 窗口的底部，然后按 Execute 按钮，会发生什么？第二个表达式究竟会不会产生一个返回值，从而第三个表达式得以被求值，最后画布消失？对于依照原来的诀窍设计的函数，这种情况可能发生吗？

习题 25.1.2 开发 *move-balls*，该函数读入一个小球的表，移动每一个球，直到它们全部出界为止。**提示**：编写这个函数最好的方法是使用本书第四部分中的 filter、andmap 以及类似的抽象函数。

25.2 快速排序

生成递归的一个经典例子是霍尔快速排序算法。与第 12.2 节中的 *sort* 一样，*qsort* 是一个排序函数，它读入一个数表，返回升序排列的表，表中包含相同的数。两个函数之间的区别是，*sort* 是基于结构递归的，而 *qsort* 是基于生成递归的。

生成步骤的基本思想是一种古老的策略：分而治之。换一种说法，我们把非平凡可解问题的实例分解为相关的两个小问题，分别解决这两个小问题，再把它们的解结合起来，从而得到原问题的解。以 *qsort* 为例，先把数表分为两个表：一个表包含所有严格小于第一个元素的元素，而另一个表包含所有严格大于第一个元素的元素；然后，对这两个（小的）表使用同样的方法排序；一旦这两个小表的排序完成了，只需简单地把它们连接起来。由于第一个元素的特殊角色，我们把它称为关键元素。

为了更好地理解这个过程，我们用手工来计算其中的一个步骤。假设输入是：

```
(list 11 8 14 7)
```

那么关键元素就是 11。把这个表分为大于和小于 11 的两部分，可得如下两个表：

```
(list 8 7)
```

和

```
(list 14)
```

这第二个表已经是按升序排列的了；对第一个表排序，可得**(list 7 8)**。这样，我们就把原来的表分成了三个部分：

1. (list 7 8)，由较小的数组成的有序表；
2. 11；
3. (list 14)，由较大的数组成的有序表。

只需把这三个部分连接起来，就可以得到原表的排序结果：(list 7 8 11 14)。

我们还有没有说明 *qsort* 如何知道自己应该停止计算。既然它是一个基于生成递归的函数，一般的答案就是，当排序问题变成一个平凡可解问题时，它就停止。显然，对 *qsort* 来说，empty 是一种平凡的输入，因为它唯一的排序结果就是 empty。到此为止，整个答案就完整了：我们会在下一章中再讨论（什么是平凡可解问题）这个问题。

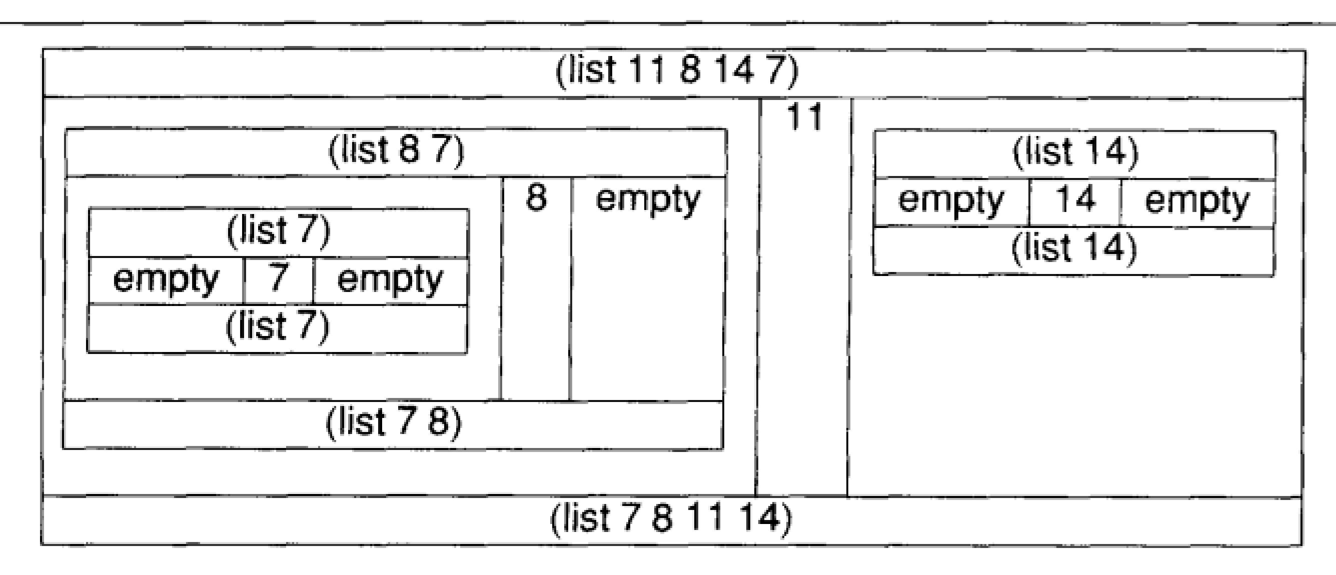


图 25.2 *quick-sort* 一个表格式的说明

图 25.2 以表格的形式给出了对**(list 11 8 14 7)**排序的全部过程。每个方框都包含三个部分：

要排序的表		
对比关键元素小的部分排序	关 键 元 素	对比关键元素大的部分排序
有 序 表		

顶层显示了我们要对之排序的表，而底层给出了排序的结果。中间的三栏显示了对两个分割部分的排序及关键元素。

习题

习题 25.2.1 模拟对(list 11 9 2 18 12 14 4 1)排序的全部 qsort 步骤。

理解了生成步骤之后，现在可以把对这个过程的描述翻译成 Scheme。该描述说明 *qsort* 区分两种情况，如果输入是 empty，它返回 empty；否则，它执行生成递归。这表明我们需要一个 cond 表达式：

```
;; quick-sort : (listof number) -> (listof number)
;; 构造一个升序的数表，由 alon 中所有的数组成
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else ...]))
```

在第一种情况下，答案是已知的。对于第二种情况，*qsort* 的输入不是 empty，算法使用第一个元素把表的其余部分分成两个子表：一个表由所有小于关键元素的元素组成，另一个表由所有大于关键元素的元素组成。

既然表的其余部分的长度是未知的，我们把分割表的任务交给两个辅助函数处理：*smaller-items* 和 *larger-items*。它们处理表，分别选出小于以及大于第一个元素的元素。因此，这两个辅助函数都使用两个参数：数表和数。当然，这两个函数是结构递归的，图 25.3 给出了它们的定义。

```
;; quick-sort : (listof number) -> (listof number)
;; 构造一个升序的数表，由 alon 中所有数组成
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else (append
            (quick-sort (smaller-items alon (first alon)))
            (list (first alon))
            (quick-sort (larger-items alon (first alon))))]))

;; larger-items : (listof number) number -> (listof number)
;; 构造一个表，由 alen 中所有大于 threshold 的数组成
(define (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else (if (> (first alon) threshold)
              (cons (first alon) (larger-items (rest alon) threshold))
              (larger-items (rest alon) threshold))]))

;; smaller-items : (listof number) number -> (listof number)
;; 构造一个表，由 alen 中所有小于 threshold 的数组成
(define (smaller-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else (if (< (first alon) threshold)
              (cons (first alon) (smaller-items (rest alon) threshold))
              (smaller-items (rest alon) threshold))]))
```

图 25.3　快速排序算法

两个子表分别再使用 *quick-sort* 排序，这就是递归，更确切地说，我们使用如下两个表达式：

1．(*quick-sort* (*smaller-items alon* (first *alon*)))，对由比关键元素小的元素组成的表排序；

2．(*quick-sort* (*larger-items alon* (first *alon*)))，对由比关键元素大的元素组成的表排序。

一旦得到了这两个表的排序结果，我们需要一个函数，把这两个表和关键元素连接起来。Scheme 的 append 函数可以实现这一功能：

```
(append (quick-sort (smaller-items alon (first alon)))
        (list (first alon))
        (quick-sort (larger-items alon (first alon))))
```

显然，表 1 中所有的元素都比关键元素小，而关键元素又比表 2 中所有的元素小，所以这样得到的结果就是有序表。图 25.3 给出了完整的函数，其中包括了 *quick-sort*、*smaller-items* 和 *larger-items* 的定义。

我们再来看看开始那个例子的手工计算过程：

```
  (quick-sort (list 11 8 14 7))
= (append (quick-sort (list 8 7))
          (list 11)
          (quick-sort (list 14)))
= (append (append (quick-sort (list 7))
                  (list 8)
                  (quick-sort empty))
          (list 11)
          (quick-sort (list 14)))
```

```
= (append (append (append (quick-sort empty)
                          (list 7)
                         (quick-sort empty))
                  (list 8)
                  (quick-sort empty))
          (list 11)
          (quick-sort (list 14)))
= (append (append (append empty
                         (list 7)
                          empty)
            (list 8)
             empty)
          (list 11)
            (quick-sort (list 14)))
= (append (append (list 7)
          (list 8)
           empty)
  (list 11)
  (quick-sort (list 14)))
= ...
```

该计算显示了排序的基本步骤，即分割、递归排序和连接。通过这个计算，我们可以看到，***quick-sort*** 实现了图 25.2 中演示的过程。

习题

习题 25.2.2　完成上述手工计算。

手工计算说明，*quick-sort* 还有另外一种平凡情况。如果 *quick-sort* 读入的是只有一个元素的表，它返回同样的表。毕竟，对只包含一个元素的表排序的结果只能是该表自身。

修改 *quick-sort* 的定义，以利用这个观察结果。

再一次手工计算同样的例子。修改后的算法可以节省多少步骤？

习题 25.2.3　虽然在大多数情况下，*quick-sort* 能够很快缩短表的长度，但是对于较短的表来说，它运行起来相当慢。所以，人们通常先使用 *quick-sort* 缩短表的长度，再使用另一种排序函数来处理足够小的表。

开发 *quick-sort* 的变体，如果读入的表的长度比某个值小，就使用第 12.2 节中的 *sort* 函数。

习题 25.2.4　如果 *quick-sort* 读入的表中包含重复的数，该算法会返回一个严格地比输入短的表，为什么？改正这个问题，使得输出总是与输入一样长。

习题 25.2.5　使用 filter 函数，只用一行代码定义 *smaller-items* 和 *larger-items*。

习题 25.2.6　开发 *quick-sort* 的变体，其中只使用一种比较函数，比方说，只使用<。分割表时，把给定的表 *alon* 分为两个表，一个表包含 *alon* 中所有小于(first *alon*)的元素，另一个表包含所有不小于(first *alon*)的元素。

使用 local 把这些函数连接成单个函数。抽象这个新函数，让它读入一个表和一个比较函数：

```
;; general-quick-sort : (X X -> bool) (list X) -> (list X)
(define (general-quick-sort a-predicate a-list) ...)
```

设计算法



初一看，算法 *move-until-out* 和 *quick-sort* 几乎没有什么共同点。一个算法处理结构体，另一个处理表；一个算法在生成步骤中建立一个结构体；另一个把表分为三个部分，对其中的两个进行递归。简而言之，对生成递归的这两个例子的比较表明，设计算法是一种专门的行为，不可能总结出一个通用的设计诀窍。不过事情并不完全是这样的。

第一，尽管我们说算法是解决问题的过程，但它们仍然是读入并返回数据的函数。换一种说法，我们仍然选用数据来表示问题，并且，如果要理解算法的过程，必须理解数据的本质。第二，我们用数据来描述问题，例如，“建立一个新的结构体”或者是“分割数表”。第三，我们总是把输入数据分成平凡可解的和非平凡可解的。第四，设计算法的关键是问题的生成。虽然如何产生一个新问题可能独立于数据表示法，但它必然是由问题的表示法实现的。第五，一旦生成的问题被解决了，完整的解必定是由某些值组合而成的。

我们来检查结构化设计诀窍的六个一般阶段：

数据分析和设计：数据表示法的选择常常会影响我们思考问题。有时候，过程描述就指定了表示法。在其他情况下，我们可以研究其他的表示法，而且这样做也是值得的。在许多情况下，我们必须分析并定义数据集。

合约、用途说明及头部：我们还需要函数的合约、定义的头部以及用途描述。既然生成步骤与数据定义的结构没有任何的关系，用途说明不仅要指明函数做了什么，还应该包括一个注释，用普通的术语解释它是如何工作的。

例子：在以前的设计诀窍中，例子仅仅说明对于某些给定的输入，函数应该产生的输出。对于算法来说，例子应当阐明，对于给定的输入，算法是怎样运行的。这将帮助我们设计算法，也帮助读者理解算法。对于像 *move-until-out* 这样的函数，其过程是显而易见的，并不需要多少文字来说明。对于其他的函数，包括 *quick-sort*，其过程中的生成步骤依赖于一种非平凡思想，所以其解释需要较好的例子，例如图 25.2 中的例子。

模板：讨论表明，算法通用的模板是：

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
       ... problem ...
       (generative-recursive-fun (generate-problem-1 problem))
       ⋮
       (generative-recursive-fun (generate-problem-n problem)))]))
```

定义：当然，模板只是一个起提示作用的设计蓝图，而不是最终的函数形态。模板中的每一个函数只是提醒我们考虑如下四个关键问题：

1. 什么是平凡可解问题？

2. 相应的解是什么？

3. 如何生成新的、比原来的问题更容易解的问题？是生成一个新问题，还是若干个？

4. 原问题的解是不是就是（某一个）新问题的解？或者，是否需要把新问题的解连接起来，从而产生原问题的解？如果是这样，还需要原问题中的任何信息吗？

要定义算法，必须用所选用的数据表示法表达这四个问题的答案。

测试：一旦得到了完整的函数，还必须对它进行测试。跟以前一样，测试的目标是发现函数的问题，并予以消除。回忆一下，测试并不能证实函数对所有可能的输入都能正确工作。再回忆一下，最好的测试方法是，用布尔表达式来表示测试，并自动地比较期望值是否与函数的返回值相同。

习题

习题 26.0.1　对于小球在画布上运动直至出界的建模问题，给出四个关键问题的非正式答案。

习题 26.0.2　对于 *quick-sort* 问题，给出四个关键问题非正式的答案。其中有多少个 *generate-problem* 的实例？

26.1 终　止

不幸的是，标准的设计诀窍并不足以用来设计算法。到目前为止，对于任何合法的输入，函数总是能够产生输出。换言之，计算的过程总会停止。毕竟，依据诀窍的本质，每个自然递归都直接使用输入中的某个部分，而不是整个输入。因为数据是以层次的形式构造的，这意味着在递归的每一个阶段，输入都会缩短。因此函数迟早会读入一个不可分割的数据，从而停止。

对于基于生成递归的函数来说，这就不再是真的了。内部的递归不再读入输入的某个直接部分，而是某种新的、由输入生成的数据。正如习题 25.1.1 所示的，这个步骤可能会反复产生同样的输入，从而阻碍计算，永远也不会产生返回值。这时，我们说程序形成一个环或是说程序进入无限循环。

另外，在从过程描述到函数定义的翻译过程中，即使是最轻微的错误都可能导致无限循环。我们可以通过一个例子来说明这个问题。考虑如下的 *smaller-items* 定义，它是 *quick-sort* 两个“问题发生器”中的一个。

```
;;smaller-items: (listof number) number -> (listof number)
;;构造一个表，由 alon 中所有小于等于 threshold 的数组成
(define (smaller-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else (if (<= (first alon) threshold)
              (cons (first alon) (smaller-items (rest alon) threshold))
              (smaller-items (rest alon) threshold))]))
```

这里没有使用<而是用<=来比较两个数。结果，当函数作用于**(list 5)**和 **5** 时，它会返回**(list 5)**。

更糟糕的是，如果把图 25.3 中的 *quick-sort* 与这个新的 *smaller-items* 一起使用，对于输入**(list 5)**，它不会产生任何输出：

```
  (quick-sort (list 5))
= (append (quick-sort (smaller-items 5 (list 5)))
          (list 5)
```

```
             (quick-sort (larger-items 5 (list 5))))
  = (append (quick-sort (list 5))
             (list 5)
             (quick-sort (larger-items 5 (list 5))))
```

第一次递归调用要求 *quick-sort* 求出对(list 5)的排序——但是那正是我们原来要解决的问题。既然这是一个循环的计算，(*quick-sort*(list 5))永远不会产生返回值。更一般地说，没有什么能够保证递归调用的输入比原来的输入更容易求解。

这个例子给我们的教训是，在设计算法的诀窍中，还需要另外一个步骤：终止论证。这个步骤解释对于每种输入为什么程序产生输出，以及函数是怎样实现这种思想的；或者给出警告，说明在什么情况下程序可能不会终止。对于 *quick-sort* 来说，论证可能是这样的：

在每一个步骤中，*quick-sort* 使用 *smaller-items* 和 *larger-items* 把表分为两个部分。这两个函数都给出一个比输入（第二个参数）更短的表。因此，*quick-sort* 的每一步递归调用都读入一个比原来输入严格更短的表。最终，*quick-sort* 会读入 empty，并返回 empty。

如果没有这样一个论证，我们就认为算法是不完整的。

一个良好的终止论证有时可能会揭示出其他的终止情况。例如，对于任意的 *N*，(*smaller-items N* (list *N*))和(*larger-items N* (list *N*))总是给出 empty。所以我们知道，对于(list *N*)，*quick-sort* 的答案就是(list *N*)[1]。要把这条知识加入到 *quick-sort* 之中，可简单地添上一个 cond 子句：

```
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [(empty? (rest alon)) alon]
    [else (append
           (quick-sort (smaller-items alon (first alon)))
           (list (first alon))
           (quick-sort (larger-items alon (first alon))))]))
```

其中条件(empty?(rest *alon*))判断 *alon* 是否只包含一个元素。

阶段	目标	任务
例子	通过例子描述输入—输出关系以及计算过程	• 构造并显示平凡可解问题的例子。 • 构造并显示需要处理的例子。 • 举例说明如何完整地处理例子。
主体	定义一个算法	• 给出平凡可解问题的检验标准。 • 给出平凡可解问题的答案。 • 指定怎样由给定的问题生成新的问题，其中可能要使用辅助函数。 • 指定怎样把这些问题的解连接成原问题的解。
⋮	⋮	⋮
终止	证明算法对于任何可能的输入都会终止	说明递归调用的输入要比原输入短

图 26.1 设计算法

[1] 当然，我们也可以认为，单元素表的排序结果就是该表自身，这就是习题 25.2.2 的基础。

图 26.1 总结了有关算法设计的建议。其中的省略号表示算法设计还需要一个新的步骤：终止论证。请结合前述章节中的表格，学习这个表格。

习题

习题 26.1.1 定义函数 *tabulate-div*，它读入数 *n*，给出其所有的因子，从 1 开始，到 *n* 结束。如果 *n* 除以 *d* 的余数是 0，即(=(remainder *n d*)0)为真，那么数 *d* 是数 *n* 的因子。任何一个数，其最小的因子是 1，最大的因子是它自己。

习题 26.1.2 开发函数 *merge-sort*，对数表排序（升序排列），使用如下两个辅助函数：

1．第一个辅助函数 *make-singles*，它使用给定的表中的数构造一个单元素表的表。例如：

```
(equal? (make-singles (list 2 5 9 3))
        (list (list 2) (list 5) (list 9) (list 3)))
```

2．第二个辅助函数 *merge-all-neighbors*，它合并相邻的两个表。更精确地说，它读入一个数表的表，合并相邻的表。例如：

```
(equal? (merge-all-neighbors (list (list 2) (list 5) (list 9) (list 3)))
            (list (list 2 5) (list 3 9)))
    (equal? (merge-all-neighbors (list (list 2 5) (list 3 9)))
            (list (list 2 3 5 9)))
```

一般来说，这个函数生成一个大概是输入表一半长的表。为什么输出的表并不总是输入表的一半长呢？

确保这两个函数的开发相互独立。

函数 *merge-sort* 首先使用 *make-singles* 把输入表分割为由单元素表组成的表；然后不断地使用 *merge-all-neighbors* 合并表，并且每次都使用 *sort* 函数对每个小表排序，直至最终得到单一的表；最后这个表就是 *merge-sort* 的返回值。

26.2 结构递归与生成递归的比较

算法的模板是这样的一般，以至于它还覆盖了基于结构递归的函数。考虑包含一个终止子句以及一个生成步骤的算法：

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
      (combine-solutions
        problem
         (generative-recursive-fun (generate-problem problem)))]))
```

如果我们把 *trivially-solvable?*替换成 empty?，把 *generate-problem* 替换成 rest，算法的轮廓就变成了表处理函数的模板：

```
(define (generative-recursive-fun problem)
  (cond
```

```
[(empty? problem) (determine-solution problem)]
[else
  (combine-solutions
problem
(generative-recursive-fun (rest problem)))]))
```

习题

习题 26.2.1 定义 *determine-solution* 和 *combine-solutions*，使得函数 *generative-recursive-fun* 计算输入的长度。

这就提出了一个问题，基于结构的递归和基于生成的递归之间有没有区别。回答是"根据情况而定"。当然，我们可以说所有使用结构递归的函数都只是生成递归的特例。但是，如果我们想要理解设计函数的过程，这种"万物皆相等"的态度对我们是没有任何帮助的。这样想会混淆两种类型的、由不同的方法得到的、并且结果不同的函数。一种设计方法基本上只依赖于系统的数据分析；另一种设计方法需要对问题解决过程本身深刻的——通常是数学上的——洞察力。一种设计方法引导程序员给出自然终止的函数；另一种设计方法需要程序进行终止论证。

简单地观察某个函数的定义，我们很快就可以判断出它是用了结构递归还是生成递归。所有自我引用的结构递归函数总是使用当前输入的某个直接部分作为下一步的输入，对它进一步处理。例如，对于某个用 **cons** 构造的表，其直接成分就是表的 **first** 元素和 **rest** 部分。因此，如果某个函数的参数是一个普通的表，而且它并不使用表的其余部分进行递归，那么这个函数定义就不是结构递归的，而是生成递归的。或者说，严格意义上的递归算法使用重新生成的输入作递归调用的参数，该参数可能包含输入的成分，也可能不包含输入的成分。无论是哪种情况，新的数据代表的问题与原来给定的问题不同，但仍然属于同一个类型。

26.3 做出选择

用户并不能区分 *sort* 和 *quick-sort*。这两个函数都读入一个数表，返回一个升序排列的数表，其中包含了相同的数。对于观察者来说，这两个函数是完全等价的[1]。这就提出了一个问题，程序员应当提供两者中的哪一个。更一般地说，如果既可以开发使用结构递归的函数，也能够开发等价的使用生成递归的函数，我们该开发哪一个？

为了能更好地理解这种选择，我们来讨论另一个例子，生成递归在数学方面的经典例子：寻找两个正自然数最大公因子的问题。所有的正自然数（即正整数）都至少有一个相同的因子：1。有时候，这就是唯一的公因子。例如，2 和 3 唯一的公因子就是 1，因为它们其他的因子分别就是 2 和 3 自身。此外，6 和 25 都有好几个因子：

1. 6 可以被 1、2、3 及 6 整除；
2. 25 可以被 1、5 及 25 整除。

尽管如此，25 和 6 的最大公因子还是 1。与此相反，18 和 24 有许多公因子：

1. 18 可以被 1、2、3、6、9 及 18 整除；
2. 24 可以被 1、2、3、4、6、8、12 及 24 整除。

它们的最大公因子是 6。

遵照设计诀窍，我们从合约、用途说明和头部开始：

[1] 在学习程序设计语言和它们的含义时，函数或者表达式在观察上等价这个概念起着非常重要的作用。

```
;;gcd : N[>= 1] N[>= 1] -> N
;;寻找 n 和 m 的最大公因子
(define (gcd n m)
  ...)
```

合约精确指定了输入：大于等于 **1** 的自然数（不能是 **0**）。

现在，我们需要做出决定，是基于结构递归进行设计，还是基于生成递归进行设计。既然答案不是很明显，我们两者都开发。对于结构递归来说，我们必须考虑函数应该处理哪个输入：*n*，*m*，还是两者都处理。稍加思考表明，我们真正需要的是这样一个函数，它从两个数中较小的一个开始，输出第一个小于或等于这个数的、能同时整除 *n* 和 *m* 的数。

```
;;gcd-structural : N[>= 1] N[>= 1] -> N
;;寻找 n 和 m 的最大公因子
;;结构递归，使用 N[>= 1]的数据定义
(define (gcd-structural n m)
  (local ((define (first-divisior-<= i)
            (cond
              [(= i 1) 1]
              [else (cond
                      [(and (= (remainder n i) 0)
                            (= (remainder m i) 0))
                       i]
                      [else (first-divisior-<= (- i 1))])])))
    (first-divisior-<= (min m n))))
```

图 26.2　通过结构递归寻找最大公因子

我们使用 local 定义一个合适的辅助函数：参见图 26.2。条件“整除”写成代码就是(=(remainder *n i*)0)以及(=(remainder *m i*)0)，它们保证了用 *n* 和 *m* 除以 *i* 都没有余数。用一些例子测试 *gcd-structural*，结果表明它能够找到期望的解。

虽然 *gcd-structural* 的设计相当简单易懂，但它也很幼稚。它只是测试每一个数，看它能否同时整除 *n* 和 *m*，然后返回第一个这样的数。对于比较小的自然数，这个过程可以工作。然而，考虑下面这个例子：

```
(gcd-structural 101135853 45014640)
```

其结果是 177，要得到这个结果，*gcd-structural* 必须要比较 101135676（即 101135853－177）个数。这是非常巨大的工作量，即使是相当快的计算机都需要好几分钟时间来完成此项工作。

习题

习题 26.3.1　把 *gcd-structural* 的定义输入到 Definitions 窗口之中，然后在 Interactions 窗口中计算(time(*gcd-structural* 101135853 45014640))。

提示：在测试 *gcd-structural* 之后，使用 Full Scheme 语言（不用调试）对之进行性能测试，比起较低的 Scheme 语言级别，这将更快地计算表达式，但是提供更少的保护。为了方便阅读，把(*require-library* "core.ss")添加到定义窗口的顶部。

数学家们在很久以前就认识到“结构化的算法”缺乏效率，此后，他们更深入地研究寻找最大公因子问题。最后，他们得出的结论是，对于两个自然数 *larger* 和 *smaller*（前者大于后者）来说，它们的最大公因子与 *smaller* 和 remainder 的最大公因子相等，其中 remainder 是 *larger* 除以 *smaller* 的余数。归结成等式形式，就是：

```
  (gcd larger smaller)
= (gcd smaller (remainder larger smaller))
```

因为(remainder *larger smaller*)比 *larger* 和 *smaller* 都要小，所以等式右边的 gcd 把 *smaller* 作为它的第一个参数。

把这个结论应用于我们的例子，可得：

1. 给定的数是 18 和 24。
2. 依照数学家们得出的结论，它们的最大公因子就是 18 和 6 的最大公因子。
3. 而 18 和 6 的最大公因子就是 6 和 0 的最大公因子。

这时，我们似乎陷入了困境，因为 0 的出现不在我们的意料之中。但是，0 可以被任何数整除，所以我们已经找到了答案：6。

对例子进行完整的计算，其过程不仅解释了这种思想，还说明了什么是平凡可解的情况。当两个数中较小的那个是 0 的时候，返回值就是较大的那个数。把所有这些东西结合起来，就得到了如下定义：

```
;;gcd-generative : N[>= 1] N[>=1] -> N
;;寻找 n 和 m 的最大公因子
;;生成递归：如果(<= m n)，(gcd n m) = (gcd n (remainder m n))
(define (gcd-generative n m)
  (local ((define (clever-gcd larger smaller)
            (cond
              [(= smaller 0) larger]
              [else (clever-gcd smaller (remainder larger smaller))])))
    (clever-gcd (max m n) (min m n))))
```

local 定义引入了函数的核心：*clever-gcd*，即一个基于生成递归的函数。该函数的第一个子句比较 *smaller* 和 0，从而找出平凡可解的情况，并产生相应的解。生成步骤使用上述等式，把 *smaller* 用作 *clever-gcd* 的第一个参数，把(remainder *larger smaller*)用作第二个参数。

现在，如果用 *gcd-generative* 来处理前面那个复杂的例子：

```
(gcd-generative 101135853 45014640)
```

可以看到，答案几乎瞬间就产生了。手工计算显示，*clever-gcd* 在得到解 177 之前只递归了九次。简单说来，生成递归帮助我们快速地找到此种问题的解。

习题

习题 26.3.2　给出 *gcd-generative* 四个关键问题的非正式答案。

习题 26.3.3　定义 *gcd-generative*，然后在 Interactions 窗口中计算：

```
(time (gcd-generative 101135853 45014640))
```

手工计算(*clever-gcd* 101135853 45014640)，只需给出那些引入新的 *clever-gcd* 递归调用的步骤。

习题 26.3.4　给出 *gcd-generative* 的终止论证。

就这个例子而言，使用生成递归来开发函数是很有诱惑力的。毕竟，它能更快地给出结果！但这个判断太草率了，理由有三。第一，即使是设计得很好的算法也并不总是比等价的生成递归快。例如，只有在处理较长的表时，*quick-sort* 才比较快；对于较短的表来说，标准的 *sort* 函数会更快。更糟糕的是，一个设计得较差的算法可能会给程序的性能带来灾难。第二，使用结构递归的诀窍来设计函数总是较为简单。反过来，设计算法就需要考虑如何生成新的、更小的问题，而这个步骤通常需要深奥的数学知识。第三，阅读函数的人可以轻易地理解结构递归的函数，即使在没有太多参考资料的情况下也是如此。而要理解一个算法，必须有人给你解释生成的步骤，而且，即使有了一个合适的说明，领会算法的思想可

能还是比较困难的。

经验告诉我们，大部分的函数使用的是结构递归；只有少数函数使用生成递归。当我们遇到既可以使用结构递归，又可以使用生成递归的情况，最好的方法通常是先使用结构递归，如果这样得到的程序运行起来太慢，再探究是否可以选用生成递归。如果我们选择使用生成递归，很重要的一点是，使用好的例子来说明问题的生成，并给出正确的终止论证。

习题

习题 26.3.5　手工计算：

```
(quick-sort (list 10 6 8 9 14 12 3 11 14 16 2))
```

给出递归调用 *quick-sort* 的步骤，问共需要多少次递归调用 *quick-sort*？多少次递归调用 append？对于一个长为 *N* 的表，给出一般的答案。

手工计算：

```
(quick-sort (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14))
```

问：共需要多少次递归调用 *quick-sort*？多少次递归调用 *append*？这与本习题的第一部分相矛盾吗？

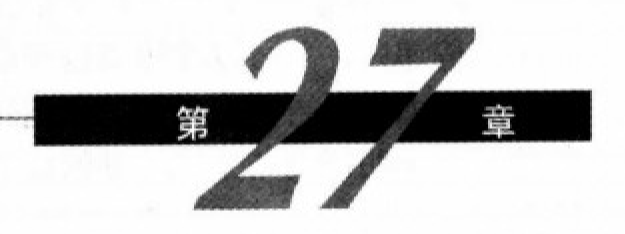
习题 26.3.6　把 *sort* 和 *quick-sort* 添加到 Definitions 窗口中。测试这两个函数，然后分别研究它们处理不同的表的速度。实验的结果（通过大量的比较）应当证实：简单的 *sort* 函数处理较短的表有优势，而 *quick-sort* 处理较长的表有优势。最后，建立 *sort-quick-sort* 函数，对于较长的表，它像 *quick-sort* 一样工作，而当表的长度短于临界值（指 *sort* 和 *quick-sort* 速度相等的值）时，它切换工作方式，像 *sort* 一样工作。

提示：（1）使用习题 26.3.5 中的思想，构造测试用的问题。（2）开发 *create-tests*，该函数随机地产生很长的测试问题。然后计算：

```
(define test-case (create-tests 10000))
(collect-garbage)
(time (sort test-case))
(collect-garbage)
(time (quick-sort test-case))
```

用 collect-garbage 来帮助 DrScheme 处理很长的表。





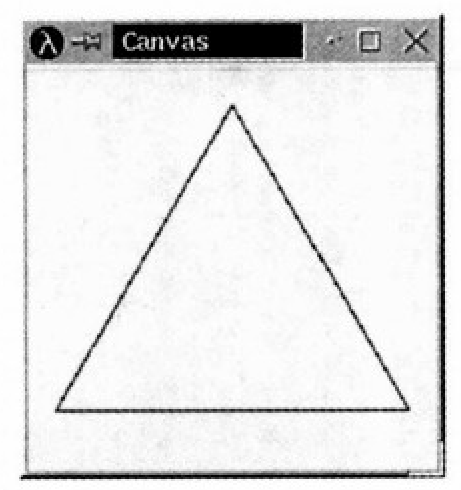
第27章 主题的变更

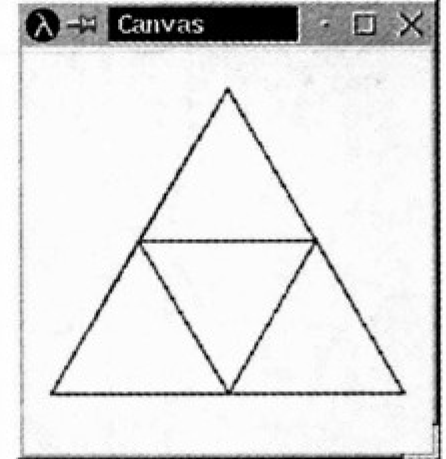
正如前两章所述，算法的设计通常从某种解决问题方法的非正式描述开始。这种描述的核心是如下两个问题：一，如何由给定的问题生成更简单的可解问题；二，更简单的问题的解是如何帮助求出原问题的解的。要找到方法，首先需要学习许多不同的例子。这一章，我们介绍几个有关生成递归设计诀窍的说明性例子。有些例子是直接从数学中得出的，而数学往往是许多问题求解方法的来源；另一些例子来自科学计算。这里的要点是要理解算法背后的思想，这样就可以在其他的环境中使用这种思想。

第一个例子是塞阿苹斯基三角形，用图形来说明生成的原理；第二个例子是“编译”中的字符序列分析过程；第三个例子是求函数的根，这是一个简单的数学上的例子，它解释了分治法的原理，许多数学过程都利用这种思想，而且理解这种思想对于应用数学特别重要。在第 27.4 节中，我们讨论另一种求根方法，它基于牛顿法。最后一节是补充练习，介绍了高斯消去法，它是解方程组的第一个步骤。

27.1 分 形

在计算几何学中，分形起着重要的作用。弗雷克（The Computational Beauty of Nature，麻省理工学院出版社，1998 年）说：“几何可以被扩展，用来描述维数为有理数的物体。这种物体被称为分形，它非常成功地描述了自然形态的丰富性和多样性。分形具有在倍数……尺寸上自我相似的结构，这意味着分形的一部分通常与其整体外表相似。”





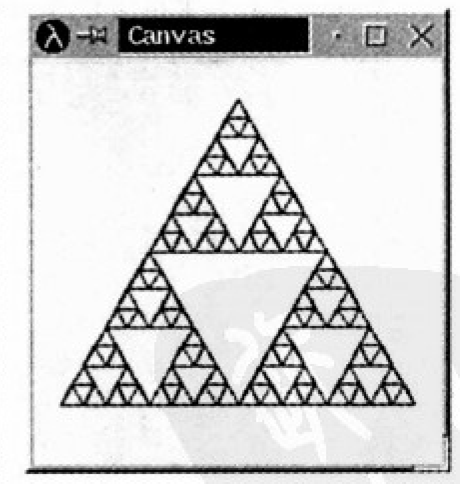


图 27.1 塞阿苹斯基三角形

图 27.1 给出了一个分形的例子，它被称为塞阿苹斯基三角形。其基本形状是一个等边三角形，正如左侧图形显示的。在右侧的图形中，我们看到该三角形被多次在最外层的三角形中复制，每次的大小各不相同。中间的图形是整个绘制过程中的一个步骤。

这中间的图形还说明了图形生成的步骤是自我相似的。给定了三角形的三个顶点，我们先绘制出这个三角形，然后计算三条边的中点。如果把这三个中点连接起来，我们就把给定的三角形分成了四个小三角形。中间的图形正描述了这种思想。反复对外侧的三个三角形进行同样的操作，而不操作中间的小

三角形，得到的结果就是塞阿苹斯基三角形。

绘制这样一套三角形的函数必须反映这个过程。它的输入数据必须代表起始三角形。如果当输入数据所描述的三角形太小了，以至于无法绘制它的时候，过程就停止。既然所有的绘图函数在完成绘制后都返回 true，绘制塞阿苹斯基三角形的函数也应当返回 true。

如果给定的三角形足够大（使得我们能够绘制它），函数必须绘制这个三角形以及嵌套在它中间的某些三角形。这里的窍门是，把分割三角形翻译成 Scheme 语句。总结这些讨论，可以得出这样一个 Scheme 定义的框架：

```
;;sierpinski : posn posn posn -> true
;;以 a、b 和 c 为顶点，绘制一个塞阿苹斯基三角形
(define (sierpinski a b c)
  (cond
    [(too-small? a b c) true]
    [else ... (draw-triangle a b c) ... ]))
```

该函数读入三个 *posn* 结构体，当任务完成后，它返回 true。cond 表达式是算法的框架。我们的任务是定义 *too-small?*和 *draw-triangle*，前者判断问题是否平凡可解，后者绘制一个三角形。另外，我们还必须添加一个 Scheme 表达式，阐述三角形的分割。

分割步骤需要函数由三个顶点给出三个中点。我们把这三个新的中点称作 *a-b*、*b-c* 和 *c-a*。这三个中点加上三个顶点可以确定四个小三角形：*a,a-b,c-a*；*b,a-b,b-c*；*c,c-a,b-c*；*a-b,b-c,c-a*。这样，如果要对某个小三角形建立塞阿苹斯基三角形，例如第一个小三角形，我们就可以使用(*sierpinski a a-b c-a*)。

因为每个中点都要被使用两次，我们使用 local 表达式把生成步骤翻译成 Scheme 语句。这个 local 表达式引入三个新的中点。local 表达式的主体包括三个对 *sierpinski* 的递归调用，以及前面所提到的 *draw-triangle* 调用。为了把这些问题的解连接起来，我们使用 and 表达式，这样就保证了所有的调用都必须成功。图 27.2 给出所有相关的定义，包括两个基于几何领域知识的小函数。

```
;; sierpinski : posn posn posn -> true
;; 以 a、b 和 c 为顶点，绘制一个塞阿苹斯基三角形
(define (sierpinski a b c)
  (cond
    [(too-small? a b c) true]
    [else
      (local ((define a-b (mid-point a b))
              (define b-c (mid-point b c))
              (define c-a (mid-point a c)))
        (and
          (draw-triangle a b c)
          (sierpinski a a-b c-a)
          (sierpinski b a-b b-c)
          (sierpinski c c-a b-c)))]))

;; mid-point : posn posn -> posn
;; 计算 a-posn 和 b-posn 的中点
(define (mid-point a-posn b-posn)
  (make-posn
    (mid (posn-x a-posn) (posn-x b-posn))
    (mid (posn-y a-posn) (posn-y b-posn))))

;; mid : number number -> number
;; 计算 x 和 y 的平均值
(define (mid x y)
  (/ (+ x y) 2))
```

图 27.2　塞阿苹斯基算法

既然 *sierpinski* 是基于生成递归的，编写代码以及测试并不是最后的步骤，我们还必须考虑为什么算法对所有合法的输入都会终止。*sierpinski* 的输入是三个点。如果输入的三角形太小，算法就会终止。每一个递归的步骤都分割三角形，使得新的三角形的边长只有原来的一半。因此，三角形的大小确实在缩小，所以 *sierpinski* 注定会返回 true。

习题

习题 27.1.1 开发下面的函数：

1. ;; draw-triangle : posn posn posn -> true
2. ;; too-small? : posn posn posn -> bool

从而完成图 27.2 中的定义。

使用教学软件包 **draw.ss** 来测试这些代码。在第一次测试完整的函数时，使用如下定义：

```
(define A (make-posn 200 0))
(define B (make-posn 27 300))
(define C (make-posn 373 300)
```

用(*start* 400 400)来建立画布。然后再使用其他的顶点和画布尺寸进行测试。

习题 27.1.2 绘制塞阿苹斯基三角形一般是从等边三角形开始的。要计算一个等边三角形的顶点，我们可以挑选一个较大的圆，选出圆周上相距 120 度的三个点。例如，它们可以是位于 0 度、120 度和 240 度的点：

```
(define CENTER (make-posn 200 200))

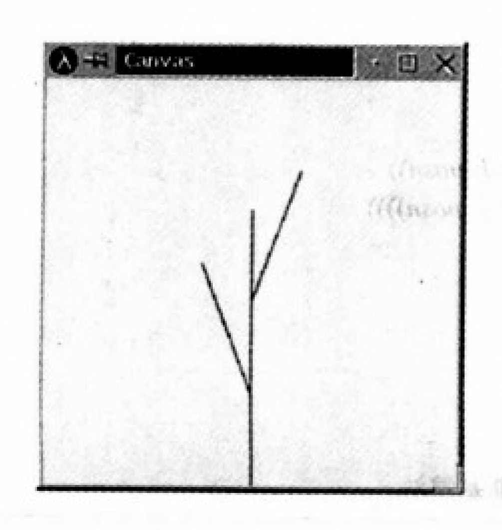
(define RADIUS 200)
;;cicrcl-pt : number -> posn
;;计算圆周上某个位置的点，该圆的圆心是如上
;;定义的 CENTER，半径是如上定义的 RADIUS
(define (circle-pt factor) ...)
(define A (circle-pt 120/360))
(define B (circle-pt 240/360))
(define C (circle-pt 360/360))
```

开发函数 circle-pt。

提示：回忆一下，DrScheme 的 sin 和 cos 函数分别计算给定弧度（不是角度）的正弦和余弦值。同时，请记住屏幕上的坐标并不是向上的，而是向下的。

习题 27.1.3 使用代表三角形的结构体，重新编写图 27.2 中的函数。然后把新的函数作用于一个三角形的表，观察其结果。

习题 27.1.4 观察如下两幅图片：

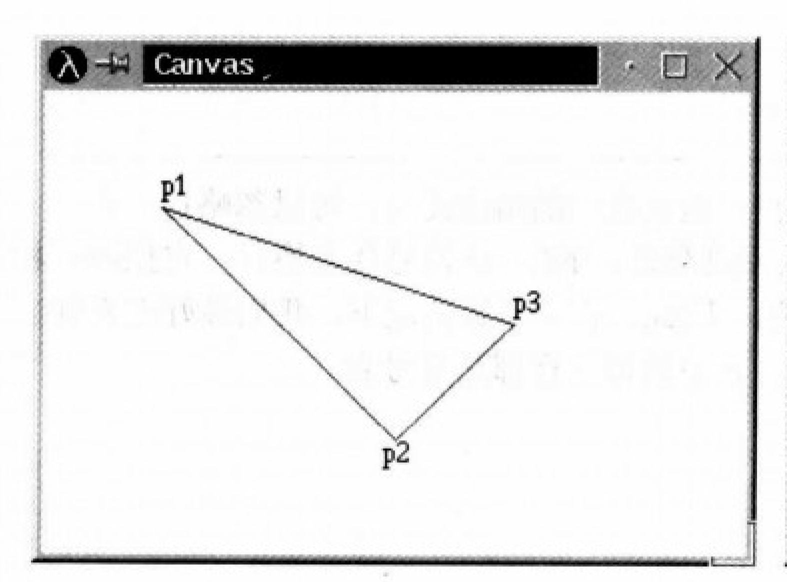


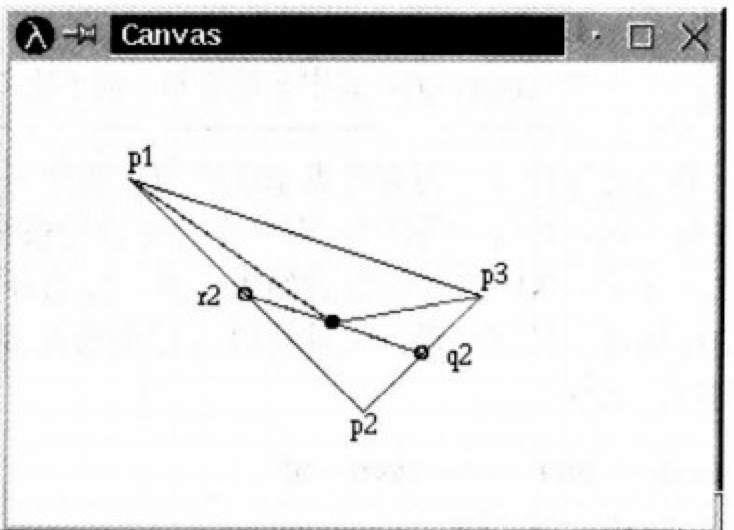


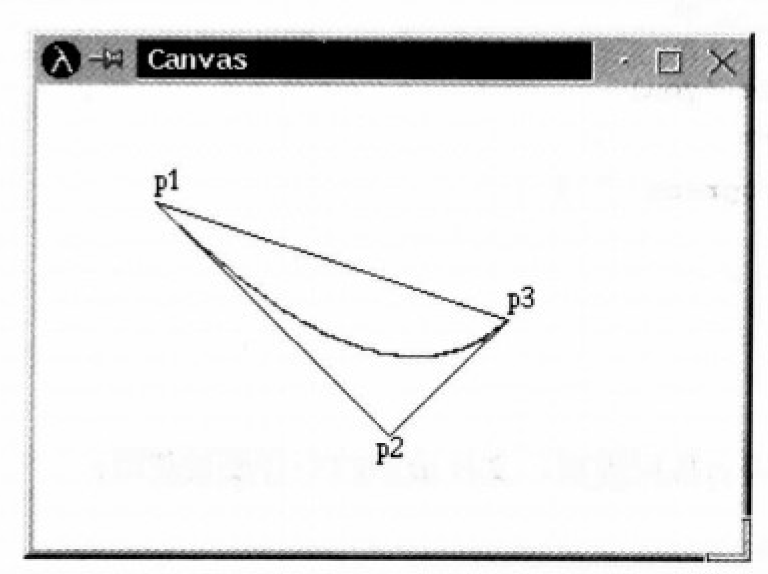
右边的图片是“萨凡纳”树，而左边的图片描述了它的基本生成步骤。左图的功能与图 27.1 中间的图类似。请开发一个函数，绘制与右图类似的树。

提示：把这个问题看作绘制一条直线，其中直线的起点、长度以及角度（即半径方向上的角度）是给定的。然后，生成步骤把这根直线三等分，使用两个等分点作为新的直线的起点。在每一步中，角度都按同样的方式改变。

习题 27.1.5　在数学和计算机图形学中，人们常常要给出连接两个点的光滑曲线。一种常见的方法是使用贝塞尔曲线。下面的一系列图片描述了绘制贝塞尔曲线的方法：







为了简单起见，我们从 *p1*、*p2* 和 *p3* 三个点开始，这三个点形成所有三幅图的最外层三角形。目标是画出连接 *p1* 和 *p3* 的光滑曲线，而且该曲线在两个端点的方向都指向 *p2*。上方的图片显示了原来的三角形；下方的图片显示了我们所希望得到的曲线。

要由一个给定的三角形绘制出这样一条曲线，我们进行如下的操作：如果三角形足够小，直接绘制该三角形，它看起来就像是一个点；否则，生成两个小三角形，如同中间的图片所示。两个外端点 *p1* 和 *p3*，依然是两个小三角形各自的外端点。两个小三角形的中间点分别是 *r2* 和 *q2*，也就是 *p1* 和 *p2* 的中点以及 *p3* 和 *p2* 的中点。*r2* 和 *q2* 的中点（在图中标记为 •）就是两个新的三角形的另一个外端点。

要测试这个函数，请使用教学软件包 draw.ss。下面是一些较为合理的测试数据：

```
(define p1 (make-posn 50 50))
(define p2 (make-posn 150 150))
(define p3 (make-posn 250 100))
```

用(*start* 300 200)来建立画布。再用其他的点进行测试。

27.2 从文件到行，从表到表的表

在第 16 章中，我们讨论了计算机文件的组织，但并没有讨论文件的本质。简单地说，我们可以把文件当作符号表：

> *file* 是下列两者之一
> 1. ***empty***。
> 2. (**cons** *s f*)，其中 *s* 是符号，而 *f* 是文件。

一个完全符合事实的文件表示法应当只包含字符，但就我们的用途而言，可以忽略这一点。

按照早期计算机的传统[1]，有一个符号总是被特别地处理：'NL。该符号代表换行，它把两行相互分开。也就是说，'NL 代表一行的结束，另一行的开始。因此，在大多数情况下，我们最好把文件当作更复杂的结构体。具体地说，文件可以用行表来表示，其中的每一行都是符号表。

例如，文件

```
(list 'how 'are 'you 'NL
  'doing '? 'NL
  'any 'progress '?)
```

应该被当作如下的三行表来处理：

```
(list (list 'how 'are 'you)
      (list 'doing '?)
      (list 'any 'progress '?))
```

类似地，文件

```
(list 'a 'b 'c 'NL
      'd 'e 'NL
      'f 'g 'h 'NL)
```

也应当用三行的表来表示，因为按照惯例，文件最后的空行要被忽略：

```
(list (list 'a 'b 'c)
      (list 'd 'e)
      (list 'f 'g 'h))
```

习题

习题 27.2.1 行表 empty、(list 'NL)和(list 'NL 'NL)分别代表什么？为什么这几个例子是重要的测试用例子？

提示：请记住文件最后的空行要被忽略。

下面是合约、用途说明和头部：

```
;; file->list-of-lines : file -> (listof (listof symbols))
;; 将文件转换成行表
(define (file->list-of-lines afile) ...)
```

把文件分割成行表的过程描述相当简单。如果文件是 empty，那么问题就是平凡可解的；在这种情

[1] 把文件分割成行的惯例最早可以追溯到 1890 年的人口普查，当时的机械计算机使用打孔的卡片。对于现代计算机来说，这种惯例根本没有用处。不幸的是，这件历史上的事件到现在还在负面地影响着现代计算机及软件技术的发展。

况下，文件并不包含任何行。否则，文件就至少包含一个符号，从而也至少包含一行。我们必须把这一行与文件的其他部分分开，接着必须把文件的其他部分也转化成行表。

用 Scheme 来描述这个过程，就是：

```
(define (file->list-of-lines afile)
  (cond
    [(empty? afile) ...]
    [else
      ... (first-line afile) ...
      ... (file->list-of-lines (remove-first-line afile)) ...]))
```

把第一行与其他行分开需要扫描一个任意长的符号表，所以我们添加两个辅助函数到函数清单中：*first-line* 和 *remove-first-line*。前一个函数收集字符，直到第一个'NL 出现，或者文件结束，也就是收集第一行所有的字符（但是不包括'NL）；后一个函数把这些符号去除，返回 *afile* 的其余部分。

这样，我们就可以轻易地填写模板中的空缺。在 *file->list-of-lines* 中，第一个子句的答案必然是 empty，因为空文件并不包含任何行。第二个子句的答案必然是用 cons 把(*first-line afile*)的值与(*file->list-of-lines* (*remove-first-line afile*))的值连接起来，因为前一个表达式求出第一行，而后一个表达式求出其余的行。另外，两个辅助函数使用结构递归的方法处理它们的输入；开发它们只是一个简单的练习。图 27.3 给出了这三个函数的定义，以及 *NEWLINE* 的变量定义。

```
;; file->list-of-lines : file -> (listof (listof symbol))
;; 把文件转换成行表
(define (file->list-of-lines afile)
  (cond
    [(empty? afile) empty]
    [else
      (cons (first-line afile)
            (file->list-of-lines (remove-first-line afile)))]))

;; first-line : file -> (listof symbol)
;; 求出 afile 的首部，直到 NEWLINE 第一次出现
(define (first-line afile)
  (cond
    [(empty? afile) empty]
    [else (cond
            [(symbol=? (first afile) NEWLINE) empty]
            [else (cons (first afile) (first-line (rest afile)))])]))

;; remove-first-line : file -> (listof symbol)
;; 求出 afile 的尾部，即第一个 NEWLINE 以后的部分
(define (remove-first-line afile)
  (cond
    [(empty? afile) empty]
    [else (cond
               [(symbol=? (first afile) NEWLINE) (rest afile)]
               [else (remove-first-line (rest afile))])]))

(define NEWLINE 'NL)
```

图 27.3　把文件转换成行表

我们来观察一下把前述的第一个文件转换成行表的过程：

```
  (file->list-of-lines (list 'a 'b 'c 'NL 'd 'e 'NL 'f 'g 'h 'NL))
= (cons (list 'a 'b 'c) (file->list-of-lines (list 'd 'e 'NL 'f 'g 'h 'NL)))
= (cons (list 'a 'b 'c)
        (cons (list 'd 'e)
              (file->list-of-lines (list 'f 'g 'h 'NL))))
= (cons (list 'a 'b 'c)
        (cons (list 'd 'e)
              (cons (list 'f 'g 'h)
                    (file->list-of-lines empty))))
= (cons (list 'a 'b 'c)
        (cons (list 'd 'e)
              (cons (list 'f 'g 'h)
                    empty)))
= (list (list 'a 'b 'c)
        (list 'd 'e)
        (list 'f 'g 'h))
```

从计算中，我们可以断定，*file->list-of-lines* 递归调用的参数几乎总不是给定文件的其余部分。更确切地说，它基本上不是给定文件的一个直接组成部分，而通常是给定文件的一个严格意义上的尾部。唯一的例外是在一行中'NL 连续出现两次。

最后，*file->list-of-lines* 的计算过程和定义说明其生成递归是简单的。每一次递归调用都使用比给定的表更短的表，因此递归的过程最终会停止，因为函数会读入 empty。

习题

习题 27.2.2 使用 local 重新组织图 27.3 中的程序。

对函数 *first-line* 和 *remove-first-line* 进行抽象，然后再使用 local 重新组织所得的程序。

习题 27.2.3 定义 *file->list-of-checks*，该函数读入一个数文件，返回一个饭店记录的表。

> file of numbers(数文件)是下列三者之一：
> 1. **empty**。
> 2. **(cons** *N F*)，其中 *N* 是自然数，*F* 是文件，
> 3. **(cons 'NL** *F*)，其中 *F* 是文件。

file->list-of-checks 的输出是一个饭店记录表，饭店记录含有两个字段：

```
(define-struct rr (table costs))
```

它们分别是桌号以及该桌的消费金额表。

例如：

```
(equal? (file->list-of-checks
          (list 1 2.30 4.00 12.50 13.50 'NL
                2 4.00 18.00 'NL
                4 2.30 12.50))
        (list (make-rr 1 (list 2.30 4.00 12.50 13.50))
              (make-rr 2 (list 4.00 18.00))
              (make-rr 4 (list 2.30 12.50))))
```

习题 27.2.4 开发函数 *create-matrix*，该函数读入数 n 以及一个包含 n^2 个数的表，生成一个包含 n 个表的表，而这些表都是 n 个数组成的表。

例子：

```
(equal? (create-matrix 2 (list 1 2 3 4))
        (list (list 1 2)
              (list 3 4)))
```

27.3 二分查找

应用数学家们把真实的世界模型化为非线性方程，然后试图解这些方程。一个最简单的例子是：

一个完全立方体的容积是 $27m^3$，它六个表面总共的表面积是多少？

根据几何知识，我们知道，如果立方体的边长是 x，它的总体积就是 x^3。因此我们需要知道 x 可能的取值，使得

$$x^3 = 27$$

一旦我们解出了这个方程，总的表面积就是 $6 \cdot x^2$。

一般来说，我们已知一个从数到数的函数 f，想要找出某个数 r，使得

$$f(t) = 0$$

r 的值被称作 f 的根。在前面所说的例子中，$f(x) = x^3 - 27$，r 的值就是立方体的边长[1]。

在过去的几个世纪中，数学家们开发了许多种方法，用来求不同类型的方程的根。这一节，我们来学习其中的一种求根方法，该方法的基础是中值定理，它是数学分析的一个早期结果。这样得出的算法是生成递归一个基于深奥数学理论的重要例子。该算法已被广大用户所熟悉，并且，在计算机科学中，它被称为二分查找算法。

中值定理告诉我们，如果 $f(a)$ 和 $f(b)$ 的符号相反，那么连续函数 f 在区间 $[a, b]$ 中至少有一个根。“连续”的意思是，这个函数的值不会“跳跃”，没有缺口，总是以“平滑的”方式延伸。该定理最好的解释方法就是某个函数的图形。图 27.4 中的函数 f 在 a 点位于 x 轴下方，而在 b 点位于 x 轴上方。它是一个连续函数，这一点从其不间断的、光滑的曲线上就可以判断出。而且，该函数确实在 a 和 b 之间的某个点与 x 轴相交。

现在请观察 a 和 b 的中点：

$$m = \frac{a+b}{2}$$

它把区间 $[a, b]$ 分割成两个等长的、更小的区间。现在我们可以计算 f 在 m 点的值，看它是大于还是小于 0。在本例中，$f(m) < 0$，所以按照中值定理，这个根会落在右边的区间中：$[m, b]$。我们的图形确认了这一点，因为图中根落在右边的半个区间中，即图 27.4 中标为“**范围 2**”的区间。

对于中值定理的抽象描述以及这个说明性的例子描述了一个求根过程。具体地说，我们多次使用二等分步骤，直到可以确定在一个很小的区间（其长度短于我们要求的精度）中，f 必然有一个根。现在，我们把这些描述翻译成 Scheme 算法，并称其为 *find-root*。

首先，我们必须精确地约定 *find-root* 的任务。它的参数是函数 f，即我们要求根的函数。另外，它

[1] 如果这个方程原来的形式是 $g(x)=h(x)$，那么我们把它转换成标准形式 $f(x)=g(x)-h(x)$。

还必须有表示区间边界的参数，用来指定我们希望它在该区间中寻找根。为了简单起见，我们说 *find-root* 还有另外两个参数：*left* 和 *right*。但是这两个参数并不是任意两个数。为了使我们的算法能够工作，我们必须假设：

```
(or (<= (f left) 0 (f right))
    (<= (f right) 0 (f left)))
```

成立。这个假设表示中值定理的条件，即函数在 *left* 和 *right* 处必须异号。

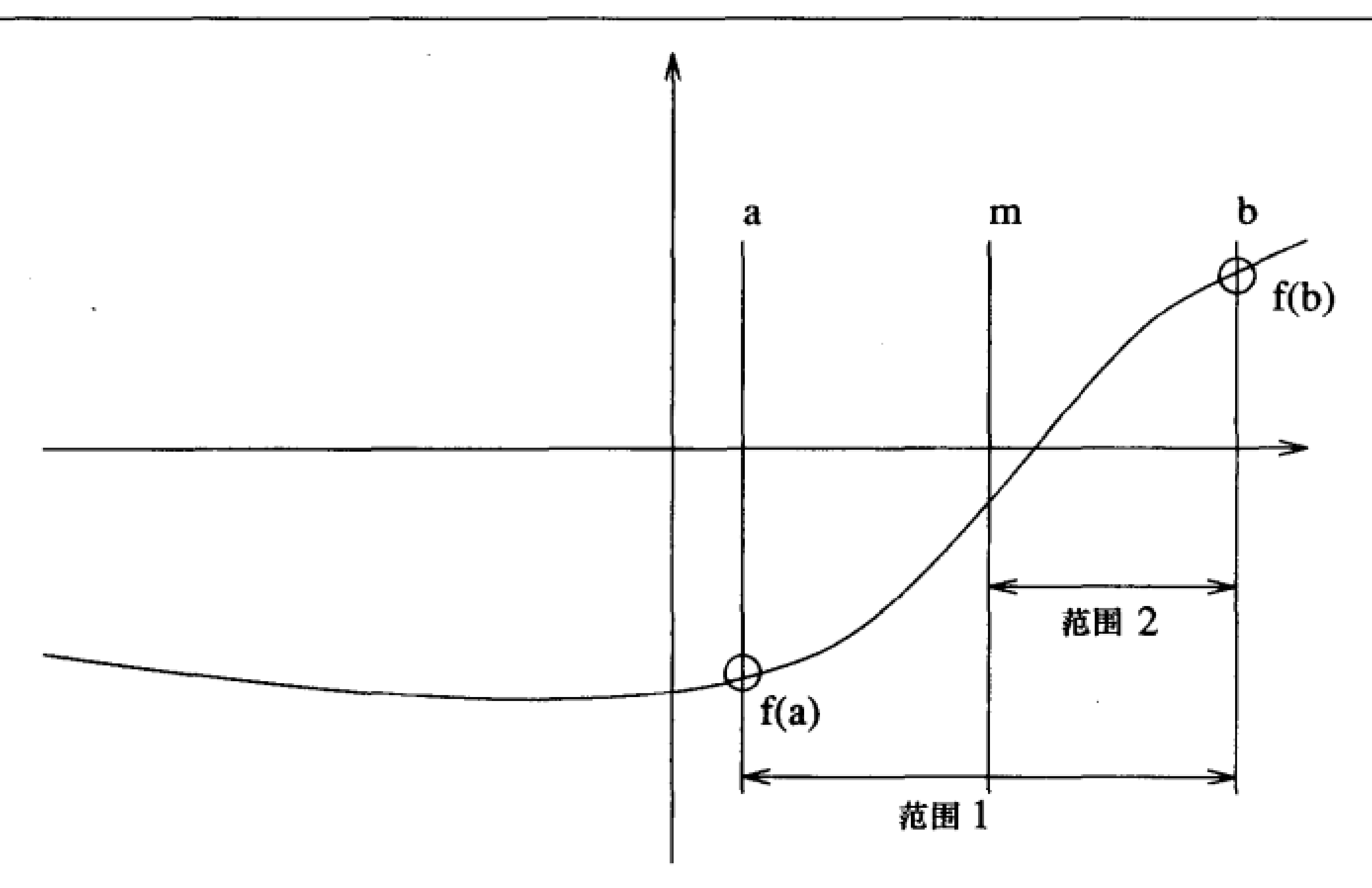


图 27.4 某个数值函数 *f*，它在区间 *[a, b]* 中有根（一阶）

按照非正式的过程描述，*find-root* 的任务是找到一个包含根的区间，并且这个区间足够小。给定的区间的大小是(- *right left*)。目前，我们暂时假定最外层定义的变量 *TOLERANCE* 表示可以容忍的最大区间。既然这样，*find-root* 可以只返回区间的两个边界中的一个，因为我们知道区间的大小最大是多少；不妨就使用左边界。

把这些讨论翻译成合约、用途描述和头部（包括关于参数的假设），其结果就是：

```
;; find-root : (number -> number) number number -> number
;; 求 R，使得 f 在[R, (+ R TOLERANCE)]中至少有一个根
;;
;;假设：(or (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
(define (find-root f left right) ...)
```

眼下，我们应该开发一个例子，说明这个函数是怎样工作的。我们已经看到了一个这样的例子；下面的习题开发第二个例子。

习题

习题 27.3.1 考虑如下的函数定义：

```
;; poly : number -> number
(define (poly x)
 (* (- x 2) (- x 4)))
```

它定义了一个二项式。我们可以手工求出它的根——2 和 4。但是，作为 *find-root* 的输入，它是一个非平凡输入，所以用它作为例子是有意义的。

请模仿基于中值定理的求根过程，求 *poly* 的根，起始的区间是[3，6]。列出如下的表格：

#step	*left*	*(f left)*	*right*	*(f right)*	*mid*	*(f mid)*
n = 1	3	-1	6.00	8.00	4.5	1.25
n = 2	3	-1	4.25	1.25	?	?

找出一个长度为.5（或更短）的区间，其中包含 *poly* 的一个根。

接下来，我们把注意力转到 *find-root* 的定义上，从 *generative-recursive-fun* 开始，考虑四个相关的问题：

1．我们需要一个条件，描述何时问题是可解的，以及相应的答案。这很简单。如果从 *left* 到 *right* 的距离小于等于 *TOLERANCE*，问题就是可解的：

```
(<= (- right left) TOLERANCE)
```

相应的返回值是 *left*。

2．我们必须给出一个表达式，生成新的 *find-root* 问题。按照非正式的过程描述，这一步需要判断中点的值，再选取下一个区间。中点会多次被使用，所以我们使用 local 表达式引入它：

```
(local ((define mid (/ (+ left right) 2)))
  ...)
```

选取区间比这要复杂得多。

再一次考虑中值定理。中值定理说，如果函数的值在给定的区间的两个端点异号，这个区间就是需要关注的候选区间。在函数的用途描述中，我们用

```
(or (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
```

来描述这个限制。因此，如果

```
(or (<= (f left) 0 (f mid)) (<= (f mid) 0 (f left)))
```

在 *left* 和 *mid* 之间的区间就是下一个被处理的区间。如果

```
(or (<= (f mid) 0 (f right)) (<= (f right) 0 (f mid)))
```

在 *mid* 和 *right* 之间的区间就是下一个被处理的区间。

简而言之，**local** 表达式的主体必须是一个条件语句：

```
(local ((define mid (/ (+ left right) 2)))
  (cond
    [(or (<= (f left) 0 (f mid)) (<= (f mid) 0 (f left)))
     (find-root left mid)]
    [(or (<= (f mid) 0 (f right)) (<= (f right) 0 (f mid)))
     (find-root mid right)]))
```

在两个子句中，我们都使用 *find-root* 继续进行查找。

图 27.5 给出了完整的函数。下面的习题要求进行测试和终止论证。

习题

习题 27.3.2　使用习题 27.3.1 中的 *poly* 测试 *find-root*。用不同的 *TOLERANCE* 值进行试验。使用第 17.8 节中的策略，用布尔值表达式表达测试。

```
;; find-root : (number -> number) number number -> number
;; 求 R，使得 f 在[R，(+ R TOLERANCE)]中至少有一个根
;;
;;假设：f 是连续平滑的
(define (find-root f left right)
  (cond
    [(<= (- right left) TOLERANCE) left]
    [else
      (local ((define mid (/ (+ left right) 2)))
        (cond
          [(<= (f mid) 0 (f right))
             (find-root mid right)]
          [else
             (find-root left mid)]))]))
```

图 27.5 求根算法 *find-root*

习题 27.3.3 假设 *find-root* 最初的参数描述了一个长度为 *S1* 的区间。在第一次递归调用 *find-root* 时，*left* 和 *right* 之间的距离是多少？第二次呢？第三次呢？经过了多少个计算步骤之后，*left* 和 *right* 之间的距离会小于或等于 *TOLERANCE*？这个问题的解答如何说明对于任何满足假设的输入，*find-root* 都会返回结果？

习题 27.3.4 对于每一个中点 *m*（除了最后一个以外），函数 *find-root* 都需要计算两次(*f m*)的值。通过对一个例子的手工计算，验证这个断言。

因为求(*f m*)的值可能要花费大量的时间，所以程序员们通常实现 *find-root* 的一个变体，避免这种重复计算。修改图 27.5 中的 *find-root*，使得它不需要重复计算(*f mid*)的值。

提示：定义一个协助函数 *find-root-aux*，使用两个额外的参数：(*f left*)的值和(*f right*)的值。

习题 27.3.5 *table*（表格）是一个函数，以一个在 0 和 *VL*（不包括）之间的自然数为参数，返回（表格中对应的）数：

```
;;g : N -> num
;;假设：i 在 0 和 VL 之间
(define (g i)
  (cond
    [(= i 0) -10]
    [(= i 1) ...]
    ...
    [(= i (- VL 1)) ...]
    [else (error 'g "is defined only between 0 and VL (exclusive)")]))
```

数 *VL* 被称为表格的长度。表格的根是表格中最接近于 0 的数。尽管我们不能读取表格的定义，但是可以通过一个搜索函数找到其根。

设计函数 *find-root-linear*，其读入一个表格和它的长度，求表格的根。使用自然数的结构归纳。这种类型的求根过程通常被称作线性查找。

如果(*t* 0)小于(*t* 1)，(*t* 1)小于(*t* 2)，以此类推，那么表格 *t* 就是单调上升的。如果表格是单调的，我们可以使用二分查找来求出其根。具体说来，可以使用二分查找找到一个长度为 1 的区间，其左边界或右边界是根的下标。开发函数 *find-root-discrete*，它读入一个表格和它的长度，求出表格的根。

提示：（1）*find-root-discrete* 的区间边界参数必须始终是自然数。考虑这会怎样影响中点的计算。（2）同样地，考虑第一个提示会怎样影响平凡可解问题情况的发现。（3）习题 27.3.3 中的终止论证还

适用吗？

如果表格的函数被定义在所有位于 0 和 1024 之间的自然数上，并且其根位于 0，使用 *find-root-discrete* 和 *find-root-lin* 来求根的区间分别需要多少次递归调用？

习题 27.3.6　我们在第 23.4 节中提到，数学家们不仅对函数的根感兴趣，还对函数在两个点之间所围起的面积感兴趣。用数学的话来说，我们对求函数在某个区间上的积分感兴趣。再一次观察图 23.1，回忆一下，我们所感兴趣的区域是由位于 *a* 和 *b* 的垂直粗线、*x* 轴以及函数的图形包围起来的。

在第 23.4 节中，我们学习了通过计算矩形面积的和的方法来近似求积分的方法。使用分而治之的策略，也可以设计一个基于生成递归计算积分的函数。粗略地说，我们把区间分成两段，分别求出每一段的积分，再把这两个值加起来。

第一步：开发算法 *integrate-dc*，按照分而治之的策略（如同 *find-root* 所使用的策略），求函数 *f* 在边界 *left* 和 *right* 之间的积分。当区间足够小时，使用矩形来近似。

尽管矩形的面积很容易计算，但矩形通常是对函数图形下方面积的一个不良近似。一种更好的几何图形是由 *a*、(*f a*)、*b* 和(*f b*)确定的梯形。它的面积是：

$$(right - left)\cdot\frac{f(right)+f(left)}{2}$$

第二步：修改 *integrate-dc*，让它使用梯形而不是矩形。

简单的分而治之方法是不经济的。考虑一个函数的图形，它在某一部分是平直的，而在另一部分是急速变化的。对于平直的部分，继续分割区间是毫无意义的，只需计算在 *a* 和 *b* 之间的梯形面积就可以了。

要发现 *f* 何时是平直的，我们可以这样改进算法。新算法不再测试区间的长度是多大，而是计算三个梯形的面积：给定的区间上的梯形以及两个平分后的区间上的梯形。假设这两者之间的差别小于

$$TOLERANCE\cdot(right - left)$$

这代表了一个高为 *TOLERANCE* 的小矩形的面积，还代表了我们的计算的误差容限。换句话说，算法判断 *f* 是否变化得太多，造成的影响是否超过了容许的误差容限，如果 *f* 没有变化过多，就停止分割区间。否则，继续使用分而治之的方法。

第三步：开发 *integrate-adaptive*，依照建议的方法求函数 *f* 在 *left* 和 *right* 之间的积分。不必讨论 *integrate-adaptive* 的终止。

自适应积分：这种算法被称为“自适应积分”，因为它自动地调整其策略。对于 *f* 是平直的部分，它只执行少量的计算；对于其他的部分，它检查很小的区间，使得误差容限相应地减小。

27.4 牛顿法

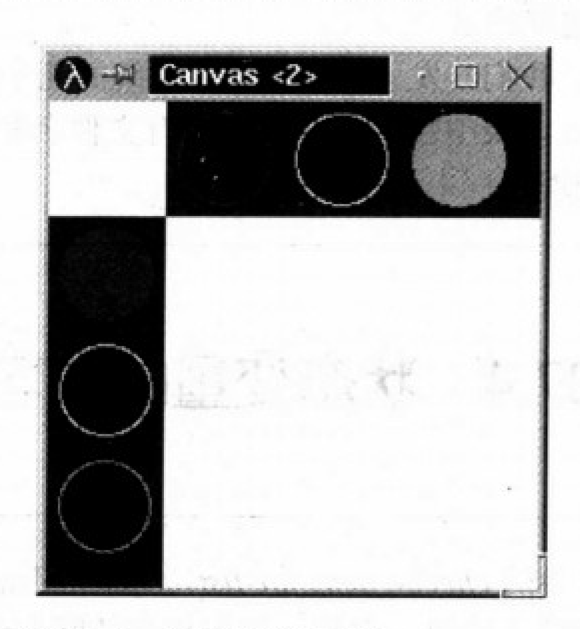
牛顿发明了另一种求函数根的方法。牛顿法使用一种近似的思想，要搜寻某个函数 *f* 的根，我们从一个猜测值，比方说 *r1* 开始，接着，考虑 *f* 在 *r1* 的切线，即通过笛卡儿坐标点（*r1*，*f*(*r1*)），并且与 *f* 在该点的斜率相同的直线。该切线是对 *f* 的一个线性近似，在大多数情况下，它有一个根，这个根比原来我们猜测的点更接近 *f* 的根。因此，通过充分多次重复这个过程，我们可以找到一个 *r*，使得(*f r*)接近于 0。

要把这个过程的描述翻译成 Scheme，我们遵循所熟悉的过程进行。这个函数——为了向其发明者表示敬意，我们称它为 *newton*——使用函数 *f* 和数 *r0* 为参数，其中 *r0* 表示当前的猜测值。如果(*f r0*)接近于 0，问题就已经被解决了。当然，接近于 0 可以被表示为(*f r0*)是一个小的正数或者是一个小的负数。因此我们把这个概念翻译成

```
(<= (abs (f r0)) TOLERANCE)
```

```
;; 效果：（1）改变 current-color，'green 变为'yellow，
;; 'yellow 变为'red，'red 变为'green
;; （2）绘制相应的交通信号灯
```

先修改函数的基本部分。在开发和测试程序时，开发如下图形显示：



使用 init-traffic-light 和 next 函数执行显示，保留其他函数。

习题 37.4.5 在第 14.4 节和第 17.7 节中，我们开发了一个 Scheme 求值程序。一个典型的 Scheme 实现还应当提供一个交互式的用户界面。在 DrScheme 中，Interactions 窗口担任这个角色。

一个交互式系统向读者提示定义和表达式，计算并返回可能的结果。定义被添加到一个知识库中；为了确定这种添加，交互式系统可能会返回一个值，比如 true。表达式使用知识库中相关的定义计算。第 17.7 节中的函数 *interpret-with-defs* 担任这个角色。

开发一个关于 *interpret-with-defs* 的交互系统，该系统至少提供两项服务：

1. *add-definition*，它把某个函数定义（的表示法）添加到系统的知识库中；
2. *evaluate*，它读入某个表达式（的表示法），使用当前知识库中相关的定义计算该表达式。

如果一个用户为某个函数 *f* 加入了两条（或更多条的）定义，只有最后一条起作用，其他定义会被忽略。

37.5 补充练习：探险

早期的电脑游戏要游戏者在危险的迷宫和洞穴中找路。游戏者从一个洞穴走到另一个洞穴，寻找财宝，遭遇各种文明，进行战斗，寻找爱情，获得能量，最终到达天国。这一节，我们使用递归的程序设计方法，设计这样游戏的一个基本部分。

我们的旅程从一个最令人恐惧的地方校园——开始。一个校园由许多建筑组成，某些建筑要比其他的更危险。每个建筑都有名字，并与其他的一些建筑相连。

游戏者总是在某一个建筑物之内。我们把这个建筑物称为当前位置。要了解有关这个位置的更多信息，游戏者可以要求得到该建筑的照片，以及相邻建筑物的表。游戏者还可以发出一个 *go* 命令，移动到一个相邻的建筑物中。

习题

习题 37.5.1 给出建筑的结构体和数据定义。在该结构体中包含一个照片字段。

校园是建筑的表。定义一个简单的校园。图 37.8 是一个例子。

使用手工计算来判断 *newton* 多快找到一个根的近似值（如果它找到了一个根的近似值）。比较 *newton* 和 *find-root* 的性能。

使用第 17.8 节中的策略，用布尔值表达式表达测试。

27.5 补充练习：高斯消去法

数学家们并不只是研究单变量方程的解，有时他们还研究线性方程组的解。下面是一个简单的线性方程组，它有 x、y 和 z 三个变量：

$$\begin{aligned} 2\cdot x+2\cdot y+3\cdot z&=10\\ 2\cdot x+5\cdot y+12\cdot z&=31 \qquad (+)\\ 4\cdot x+1\cdot y-2\cdot z&=1 \end{aligned}$$

一个方程组的解是一系列数，每个变量对应一个数，如果我们把变量替换成相应的数，每个等式两边的值就相等了。在前述的例子中，方程组的解是 $x=1$，$y=1$，$z=2$，我们可以方便地检查：

$$\begin{aligned} 2\cdot 1\cdot+2\cdot 1+3\cdot 2&=10\\ 2\cdot 1+5\cdot 1+12\cdot 2&=31\\ 4\cdot 1+1\cdot 1-2\cdot 2&=1 \end{aligned}$$

第一个方程现在变成了 10=10，第二个是 31=31，而第三个是 1=1。

最著名的求解线性方程组的方法就是高斯消去法。它由两步组成。第一步是把方程组变换成另一种形状的方程组，而保持其解不变。第二步是一次求出一个方程的解。这里我们把注意力集中在第一步之上，因为它是另一个有趣的生成递归的例子。

高斯消去算法的第一步被称为“三角剖分”，因为它产生的结果是一个三角形的方程组。与此不同，原来的方程组一般是方形的。要理解这个术语，看一看原方程组的这一种表示法：

```
(list (list  2  2   3  10)
      (list  2  5  12  31)
      (list  4  1  -2   1))
```

这种表示法抓住了方程组的本质，即变量的系数以及等式的右边。变量的名字并不起任何的作用。

在三角剖分阶段，生成步骤是从所有其他的行中减去第一行（即第一个表）。从另一行中减去一行意味着将两行中相应的元素相减。对于这个例子来说，如果我们从第二行中减去第一行，产生的结果是

```
(list (list  2  2   3  10)
      (list  0  3   9  21)
      (list  4  1  -2   1))
```

这些减法的目标是使得第一列中除了第一行以外的元素都变成 0。要使得第一列中的最后一行变成 0，我们把第三行减去两倍的第一行：

```
(list (list  2  2   3   10)
      (list  0  3   9   21)
      (list  4 -3  -8  -19))
```

换句话说，我们先把第一行中的每一个元素乘以 2，然后从最后一行中减去其结果。很容易检验原方程组的解与这个新方程组的解是相同的。

习题

习题 27.5.1 检验方程组

$$
\begin{aligned}
2\cdot x+2\cdot y+3\cdot z&=10\\
3\cdot y+9\cdot z&=21 \qquad (+)\\
-3\cdot y-8\cdot z&=-19
\end{aligned}
$$

与标记为（+）的方程组有着相同的解。

习题 27.5.2 开发 *subtract*，该函数读入两个等长的数表，从第二个表中逐元素地减去第一个表，相减足够多的次数，使得得到的结果的第一个数为 0。返回值是这个表的 rest 部分。

按照传统，我们不再写出后两个方程中开头的 0：

```
(list (list  2  2  3  10)
      (list     3  9  21)
      (list    -3 -8 -19))
```

另外，如果使用同样的过程处理剩下的方程组，生成更短的行，最终得到的方程组表示法是三角形的。

我们使用例子来研究这种思想。暂时忽略第一行，把注意力集中于剩下的方程：

```
(list (list  3  9  21)
      (list -3 -8 -19))
```

通过从第二行中减去-1 倍的第一行，（在省略开头的 0 以后）我们得到：

```
(list (list  3  9  21)
      (list     1  2))。
```

这个方程组中的其余部分（即最后一个方程）是一个单变量的方程，它不可能再被进一步简化。

把这个方程组添加到第一个方程之下，就得到：

```
(list (list  2  2  3  10)
      (list     3  9  21)      (*)
      (list        1   2))
```

正于我们所预言的，这个方程组的形状大约是个三角形，而且我们可以方便地检验，它与原来的方程组有着相同的解。

习题

习题 27.5.3 检验方程组

$$
\begin{aligned}
2\cdot x+2\cdot y+3\cdot z&=10\\
3\cdot y+9\cdot z&=21 \qquad (*)\\
1\cdot z&=2
\end{aligned}
$$

与标记为（+）的方程组有着相同的解。

习题 27.5.4 开发算法 *triangulate*，它读入一个方程组的矩形表示法，使用高斯消去法，返回三角形表示法。

不幸的是，当前的三角剖分算法有时会无法返回所需的解。考虑如下的方程组（的表示法）：

```
(list (list 2  2  3  8)
      (list    3 -2  3)
      (list 4 -2  2  4))
```

它的解是 $x=1$，$y=1$，$z=1$。

第一步应该是从第二行中减去第一行，再从第三行中减去两倍的第一行，这产生了如下的矩阵：

```
(list (list 2  3  3   8)
      (list    0 -5  -5)
      (list   -8 -4 -12))
```

下一步，算法会把注意力集中于这个矩阵的其余部分：

```
(list (list  0 -5  -5)
      (list -8 -4 -12))
```

但是该矩阵的第一个元素是 0。因为不能用 0 去除其他的数，我们遇到了困难。

要解决这个问题，我们需要使用问题所在领域内的另一条知识，即我们可以交换方程的位置，而不改变其解。当然，在交换方程的位置时，我们必须确保被移动到第一行的行的第一个元素不是 **0**。在这里，我们可以简单地交换两行：

```
(list (list -8 -4 -12)
      (list  0 -5  -5))
```

接下来，我们可以像以前一样继续，从其他行中减去足够多次的第一行。最终得到的三角形矩阵是：

```
(list (list 2  3  3   8)
      (list   -8 -4 -12)
      (list      -5  -5))
```

很容易检验，这个方程组的解仍然是 $x=1$，$y=1$，$z=1$。

习题

习题 27.5.5 修改习题 27.5.4 中的算法 *triangulate*，使得它在遇到矩阵的第一个元素是 0 的时候交换矩阵的行。

提示：DrScheme 提供了函数 *remove*。它以元素 *I* 和表 *L* 为参数，生成一个类似于 *L* 的表，但是把 *I* 的第一次出现删除。例如，

```
(equal? (remove (list 0 1) (list (list 2 1) (list 0 1)))
        (list (list 2 1)))
```

习题 27.5.6 有些方程组并没有解。作为例子，考虑如下的方程组：

$$2\cdot x+2\cdot y+2\cdot z=6$$
$$2\cdot x+2\cdot y+4\cdot z=8$$
$$2\cdot x+2\cdot y+1\cdot z=2$$

用手工来运行 *triangulate*，试着给出一个三角形矩阵。发生了什么事？修改该函数，使得它在遇到这种情况时产成错误消息。

习题 27.5.7 在得到了一个三角形的方程组，例如习题 27.5.3 中的（*）之后，我们可以解出这些方程。在这个例子中，最后一个方程表明 z 是 2。知道了这一点，通过一个替换，可以从第二个方程中

除去 z:

$$3 \cdot y+9 \cdot 2=21$$

求出 y 的值。然后重复这个步骤，替换掉第一个方程中的 y 和 z，求出 x 的值。

开发函数 *solve*，以三角形的方程组为输入，给出它的解。一个三角形的方程组的形状是：

$$\begin{array}{llllll}
(\text{list} & (\text{list} & a_{11} & \cdots & & \cdots b_1) \\
 & (\text{list} & & a_{21} & \cdots & b_2) \\
 & (\text{list} & & \vdots & \vdots & \vdots) \\
 & (\text{list} & & & a_{nn} & b_n))
\end{array}$$

其中 a_{ij} 和 b_i 是数。更确切地说，它是一个表的表，并且每一个表比前一个表少一个元素。方程组的解是一个数表。该表的最后一个元素是：

$$\frac{b_n}{a_{nn}}.$$

提示：设计 *solve* 需要先求如下问题的解。假设给定这样的一行：

```
(list 3 9 21)
```

以及一个数表，代表方程组其余部分的解：

```
(list 2)。
```

这两条数据表示：

$$3 \cdot x+9 \cdot y=21$$

以及

$$y=2$$

这反过来表明我们必须解如下的方程：

$$3 \cdot x+9 \cdot 2=21$$

开发函数 *evaluate*，该函数求出一个方程左边的其余部分的值，并从方程的右边减去这个值。等价地说，*evaluate* 读入(list 9 21)和(list 2)，返回-3，即 9•2－21。现在使用 *evaluate* 作 *solve* 的中间步骤。



回溯算法



解决问题的过程并不总是直线的。有时候，我们向某个目标推进，却因为走错了路而陷入了困境。在这种情况下，我们需要回溯，转而进入另一个可能解决问题的分支，希望这条路可以找到问题的解。算法也是如此。本章的第一节讨论一个图遍历算法，这是刚才讨论过的一种情形；第二节是一个补充练习，研究任何在下棋中使用回溯。

28.1 图的遍历

有时候，我们需要通过一个迷宫；有时候，我们希望画出一张图，表示人与人之间的关系；有时候，我们需要设计一条通过管道网的路线；有时候，我们要在国际互连网上找到一条路径，把消息从一处传送到另一处。所有的这些问题都可以用有向图来描述。

具体来说，我们有一个节点的集合以及一个边的集合。一条边表示两个节点之间的一条单向连接。观察图 28.1，黑色的圆点表示节点；它们之间的箭头表示单向连接。图中所示的图由七个节点和九条边组成。

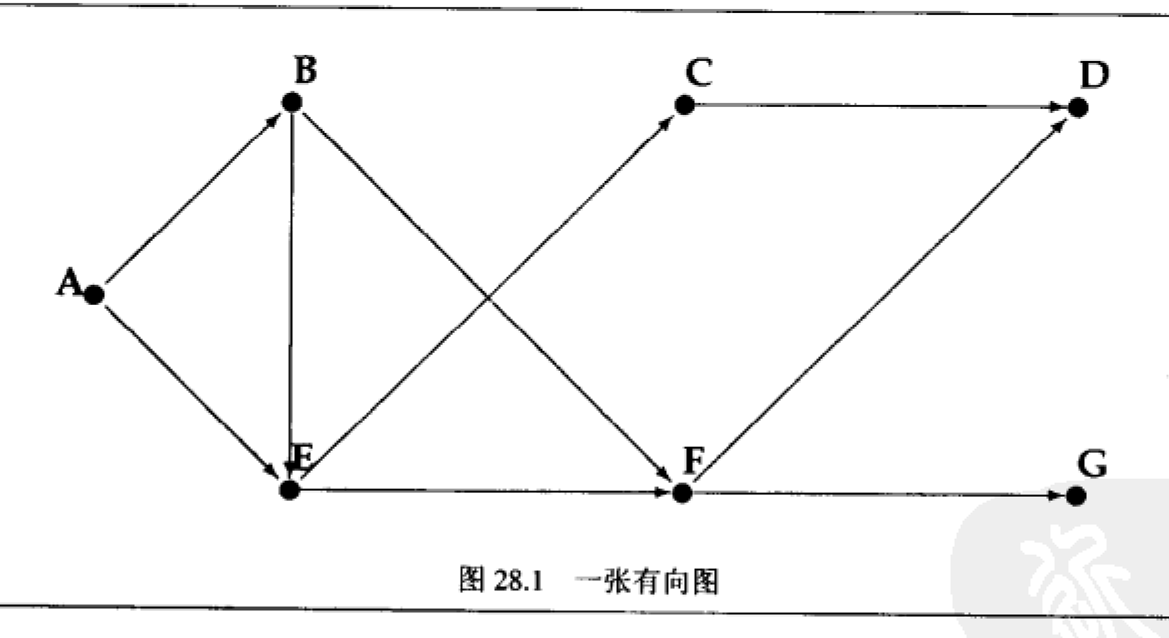


图 28.1　一张有向图

现在，假设我们要为图 28.1 中的图设计路线。例如，如果我们要从 C 走到 D，路线很简单：它由源节点 C 和目标节点 D 组成。反之，如果要从 E 走到 D，我们有两种选择：

1. 可以从 E 走到 F，然后走到 D。
2. 或者，从 E 走到 C，然后走到 D。

不过，对于某些节点，不可能在它们之间建立连接。例如，在我们的示例图中，按照箭头就不可能从 C 走到 G。

在真实的世界中，一张图可以有更多的顶点、更多条边。因此，开发在图中寻找路线的函数就是一件很自然的事。遵循一般的设计诀窍，我们从数据分析开始。下面是图 28.1 中的图的基于表的紧凑表示

法：

```
(define Graph
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

在这个表中，每一个节点对应于一个表，表由节点的名字开头，后面跟着它的邻居组成的表。例如，第二个表代表了节点 B，有两条边从它出发，分别指向 E 和 F。

习题

习题 28.1.1　使用 list 以及合适的符号，将上述定义翻译成严格意义上的表。

节点的数据定义很简单：

> *node*（节点）是符号。

给出图的数据定义，图中可以包含任意多的节点和边。该数据定义必须要包含 *Graph* 数据类型。

基于 *node* 和 *graph* 的数据定义，现在可以设计 *find-route*——在图中搜索路线的函数——的合约（草稿）：

```
;; find-route : node node graph -> (listof node)
;; 给出 G 中从 origination 到 destination 的一条路线
(define (find-route origination destination G) ...)
```

这个头部未解决的问题是返回值精确的形状。它仅说明了返回值是节点的表，但是并没有精确地说明这个表包含了哪些节点。要理解这个问题，我们必须先学习几个例子。

考虑前面所提到的第一个问题。用 Scheme 表达式的形式来描述，这个问题就是：

```
(find-route 'C 'D Graph)
```

从'C 到'D 的路线只包括两个节点：源节点和目标节点。因此，我们可以预期答案是(list 'C 'D)。当然，有些人可能认为，既然源节点和目标节点都是已知的，返回值应当是 empty。这里，我们选用第一种方案，因为它更为自然，要给出第二种形式的答案，只需对最终的函数定义作少量修改。

现在来考虑第二个问题，从'E 到'D，这个问题很有代表性。一种自然的想法是检查'E 所有的邻居，再找到一条从它们中的一个出发，到'D 的路线。在我们的示例图中，'E 有两个邻居：'C 和'F。假设这时我们还不知道最终的路线。在这种情况下，我们可以再一次检查'C 的所有邻居，找一条从它们到目标的路线。当然，'C 只有一个邻居，它就是'D。把各个阶段的结果结合起来，最终的结果就是(list 'E 'C 'D)。

最后一个例子提出一个新的问题。假设 *find-route* 得到的参数是'C、'G 和 *Graph*。在这种情况下，观察图 28.1 可知它们之间没有连通的路线。要表明路线不存在，*find-route* 应当返回一个不可能被错误理解为某条路线的值。一个合适的选择是 false，它不是一个表，并且自然地表明函数不能计算出一个合适的返回值。

这就要求合约作一点改动：

```
;; find-route : node node graph -> (listof node)或false
;; 给出 G 中一条从 origination 到 destination 的路线
```

```
;; 如果这样的路径不存在，函数返回 false
(define (find-route origination destination G) ...)
```

下一个步骤是要理解该函数的四个基本部分："平凡可解"条件、相应的解、新问题的生成，以及子问题解的结合步骤。通过对三个例子的讨论，可知它们的答案。第一，如果 *find-route* 的 *origination* 参数等于它的 *destination*，这个问题就是平凡的；相应的答案就是(**list** *destination*)。第二，如果这两个参数不相等，我们必须检查 *graph* 中 *origination* 所有的邻居，并判断是否有一条从它们中的一个到达 *destination* 的路线。

因为一个节点可以有任意多个邻居，这项任务对于一个单一的基本函数来说太复杂了。我们需要一个辅助函数。该辅助函数的任务是读入一个节点的表，然后判断在给定的图中，从表中的每一个节点出发，是否有到达目标节点的路线。换句话说，该函数是 *find-route* 的面向表的版本。我们把这个函数称为 *find-route/list*。把这些非正式的描述翻译成的合约、头部和用途说明，就是：

```
;; find-route/list : (listof node) node graph ->(listof node)或 false
;; 给出一条从 lo-originations 中的某一个节点到 destination 的路线
;; 如果这样的路径不存在，函数返回 false
(define (find-route/list lo-originations destination G) ...)
```

现在我们可以写出 *find-route* 如下的草稿：

```
(define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else ... (find-route/list (neighbors origination G) destination G) ...]))
```

函数 *neighbors* 求出从 *origination* 的邻居到 *destination* 的路线。该函数的定义是一个简单的结构递归练习。

习题

习题 28.1.2　开发函数 *neighbors*，该函数以节点 *n* 和图 *g* 为参数（参见习题 28.1.1），生成 *g* 中 *n* 的邻居表。

接下来需要考虑 *find-route/list* 生成了什么。如果它找到了一条从某个邻居出发的路线，它会生成一个从该邻居到最终目的地的路线。但是，如果没有一个邻居与目的地相连，函数返回 false。显然，*find-route* 的答案取决于 *find-route/list* 生成了什么，因此应当使用一个 cond 表达式区分 *find-route/list* 的答案。

```
(define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G)
                                     destination G)))
            (cond
              [(boolean? route) ...]
              [else ; (cons? route)
                ...]))]))
```

两种情况反映了可能得到的两种类型的答案：布尔值或是表。如果 *find-route/list* 返回 false，那么它无法找到一条从 *origination* 的邻居出发的路线，所以到达 *destination* 是完全不可能的，因此在这种情况下，答案必然是 false。反之，如果 *find-route/list* 返回一个表，答案必定是从 *origination* 到 *destination* 的

表，既然 *possible-route* 是从 *origination* 的一个邻居开始的，把 *origination* 加到 *possible-route* 的前端就可以了。

图 28.2 给出了 *find-route* 完整的定义，包括 *find-route/list* 的定义，它通过结构递归处理其第一个参数。对于表中的每一个节点，*find-route/list* 使用 *find-route* 试着生成一条路线。如果 *find-route* 确实生成了一条路线，这条路线就是答案。否则，如果 *find-route* 不能找到路线并返回 false，函数就递归。换一种说法，它回溯至当前起点(first *lo-Os*)，并试着使用表中的下一个节点。因为这个原因，*find-route* 通常被称作回溯算法。

```
;; find-route : node node graph -> (listof node) or false
;; 给出 G 中一条从 origination 到 destination 的路线
;; 如果这样的路径不存在，函数返回 false
(define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G) destination G)))
            (cond
            [(boolean? possible-route) false]
            [else (cons origination possible-route)]))]))

;; find-route/list : (listof node) node graph -> (listof node) or false
;; 给出一条从 lo-Os 中的某一个节点到 D 的路线
;; 如果这样的路径不存在，函数返回 false
(define (find-route/list lo-Os D G)
  (cond
    [(empty? lo-Os) false]
    [else (local ((define possible-route (find-route (first lo-Os) D G)))
          (cond
            [(boolean? possible-route) (find-route/list (rest lo-Os) D G)]
            [else possible-route]))]))
```

图 28.2　在图中寻找一条路线

在结构域内的回溯：第 18 章讨论了在结构域内的回溯。一个特别好的例子是习题 18.1.13，涉及到有关家谱树的回溯函数。该函数首先在家谱树的一个分枝中搜索蓝眼睛的祖先，如果这一个搜索返回 false，它就搜索树的另外一半。因为图是一般化的树，比较这两个函数是有意义的。

最后，但不是最不重要的，我们需要考虑该函数是否能在所有的情况下都生成一个答案。第二个函数，即 *find-route/list* 是一个结构递归函数，所以，假如 *find-route* 总能生成返回值，它也总能生成返回值。对于 *find-route* 来说，答案就很不明显了。如果 *find-route* 被给定图 28.1 中的图以及图中的两个节点，它总能生成一些答案。不过，对于其他的一些图，计算可能不会终止。

习题

习题 28.1.3　测试 *find-route*。用它在图 28.1 中的图中寻找一条从 A 到 G 的路线。当被要求找出一条从 C 到 G 的路线时，确定它会返回 false。

习题 28.1.4　开发函数 test-on-all-nodes，以图 g 为参数，用 g 中所有的节点对来测试 find-route。用 Graph 来测试 test-on-all-nodes。

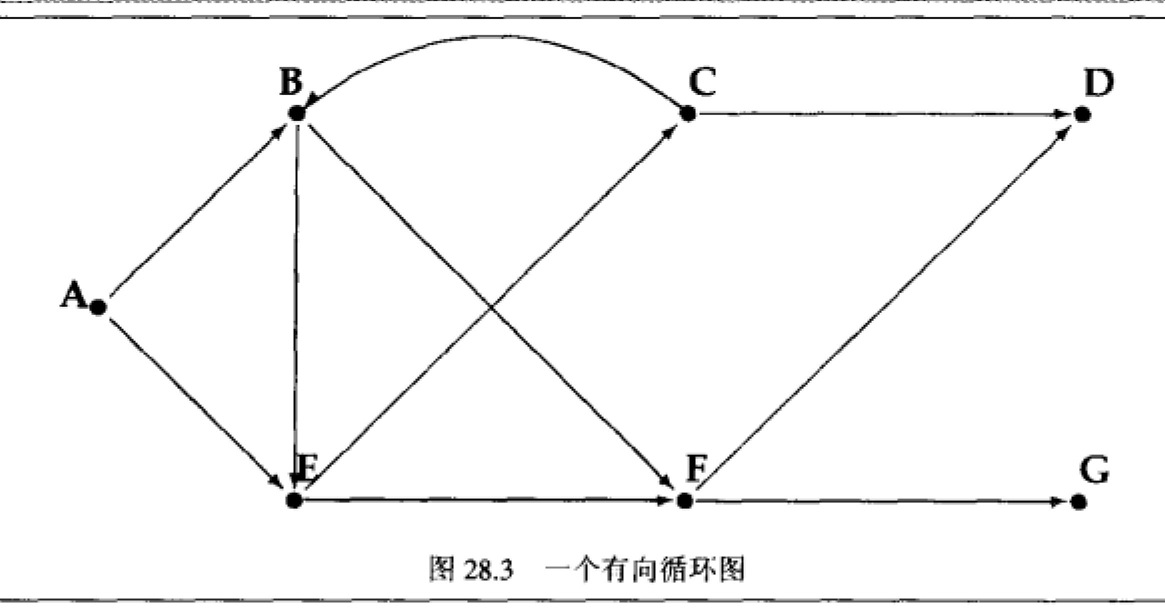


图 28.3 一个有向循环图

考虑图 28.3 中的图，它与图 28.1 中的图有着本质的区别，在这张图中，我们可以从某个节点出发，（沿着边前进）再回到同一个节点。具体来说，我们可以从 B 走到 E 再走到 C，最后返回 B。事实上，如果把 *find-route* 作用于'B，'D 以及该图的数据表示，函数不会停止。下面是手工计算的过程：

```
  (find-route 'B 'D Cyclic-graph)
= ... (find-route 'B 'D Cyclic-graph) ...
= ... (find-route/list (list 'E 'F) 'D Cyclic-graph) ...
= ... (find-route 'E 'D Cyclic-graph) ...
= ... (find-route/list (list 'C 'F) 'D Cyclic-graph) ...
= ... (find-route 'C 'D Cyclic-graph) ...
= ... (find-route/list (list 'B 'D) 'D Cyclic-graph) ...
= ... (find-route 'B 'D Cyclic-graph) ...
= ...
```

其中 *Cyclic-Graph* 代表了图 28.3 中的图的 Scheme 表示形式。手工计算说明，在对 *find-route* 和 *find-route/list* 调用七次之后，会得到一个与原来的表达式完全相同的表达式。既然对同样的输入函数会生成同样的输出、完成同样的操作，我们可以得知这个函数会永远循环，因而不会生成返回值。

总而言之，如果给定的图是无环的，*find-route* 对任意输入都会生成某种输出。毕竟，每一条路线都只能包含有限多个节点，而且路线的数量也是有限的，因此，函数要么无遗漏地检查所有从源节点出发的所有路线，要么找到一条从源节点到达目标节点的路径。然而，如果图中包含一个环，即一条从某个节点回到它自身的一条路线，那么对于某些输入，*find-route* 可能就不能得出结果。在本书的下一个部分，我们会学习一种编程技巧，即使图中存在循环，它也会帮助我们找出路线。

习题

习题 28.1.5 用'B、'C 以及图 28.3 中的图测试 *find-route*。使用第 17.8 节中的概念，用布尔值表达式的形式来表示测试。

习题 28.1.6 重写 *find-route* 程序，使它成为一个单独的函数定义。去除局部定义的函数的参数。

28.2 补充练习：皇后之间的相互攻击

国际象棋中的一个著名问题是 n 皇后问题。对这个问题来说，国际象棋棋盘是“正方形”的，比如说，是八乘八的小方格，或者是三乘三的小方格；皇后是一种棋子，它可以在横向、纵向或者斜向移动

任意多格。如果某个皇后位于这个方格中，或者它可以（一步）移动到这个方格中，我们就说这个皇后威胁这个方格。图 28.4 显示了一个例子。实心的圆点代表一个皇后，它位于第二列第六行。从圆点辐射出的粗线穿过所有被皇后威胁的方格。

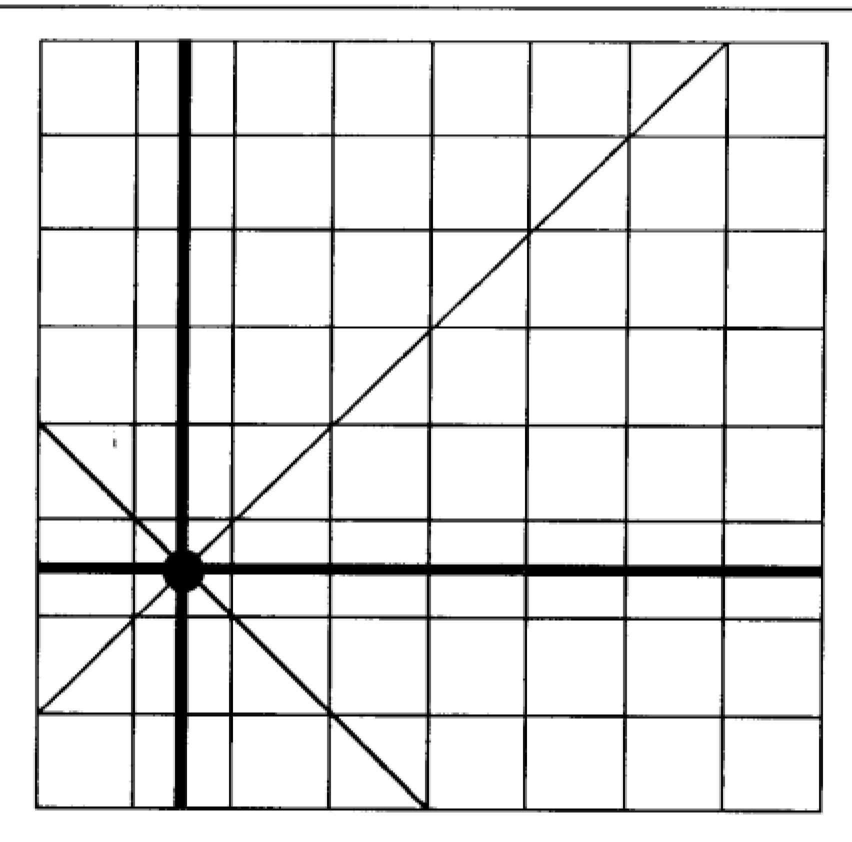


图 28.4　一个国际象棋棋盘以及一个皇后

皇后放置问题是要把八个皇后放到一个八乘八的棋盘上，使得棋盘上的皇后之间相互不构成威胁。在计算中，我们当然要一般化这个问题，问是否可以把 n 个皇后放到某个任意大小的 m 乘 m 棋盘上。

显然，在考虑设计能够解决这个问题的函数之前，我们需要一种棋盘的数据表示法以及一些基本的处理棋盘的函数。下面从一些基本的数据和函数的定义开始。

习题

习题 28.2.1　开发棋盘的数据定义。

提示：使用表。用 true 和 false 代表小方格。值 true 应当表示一个可以放入皇后的位置；false 应当表示该位置已经被皇后占据，或者被某个皇后所威胁。

接下来我们需要一个建立棋盘的函数以及一个检查特定方格的函数。仿效表的例子，我们来定义 *build-board* 和 *board-ref*。

习题

习题 28.2.2　开发如下两个关于棋盘的函数：

```
;; build-board : N (N N -> boolean) -> board
;; 建立一个大小为 n×n 的棋盘
;; 把(f i j) 填入标号为 i 和 j 的位置
(define (build-board n f) ...)

;; board-ref : board N N -> boolean
;; 访问 a-board 中标号为 i 和 j 的位置
(define (board-ref a-board i j) ...)
```

对它们进行严格的测试！使用第 17.8 节中的思想，用布尔值表达式表示测试。

除了这些一般的关于棋盘表示法的函数，我们还至少需要一个函数，用来记录问题描述中提到的“威胁”概念。

习题

习题 28.2.3 开发函数 *threatened?*，计算一个皇后能否从某个给定的位置到达棋盘上的另一位置。也就是说，该函数读入两个位置，它们以 *posn* 结构体的形式给出。如果第一个位置上的皇后可以威胁到第二个位置，函数就返回 true。

提示：这个习题把国际象棋中的 n 皇后问题转变为一个数学问题，即判断在某个给定的棋盘中，两个位置是否处于同一条线（包括横线、竖线和斜线）上。请记住，通过每一个位置的斜线有两条，它们的斜率分别是+1 和-1。

一旦有了“棋盘”的数据定义和函数，我们就可以开始处理主要任务：把一定数量的皇后放到给定的棋盘上。

习题

习题 28.2.4 开发 *placement*。这个函数读入一个自然数和一个棋盘，试着把那么多的皇后放到棋盘上。如果皇后可以被放到该棋盘上，函数就生成这样的一个棋盘；如果不能，它返回 false。



计算的代价和向量



在第 26.3 节中，我们讨论了结构递归程序与等效的生成递归程序之间的区别。比较显示生成递归要比结构递归快得多。为了证明我们的结论，我们对这两者都使用了非正式的参数——递归调用的次数，也使用了度量法——time 表达式（习题 26.3.1 和习题 26.3.3）。

尽管测量某个程序调用对于特定的输入所花费的时间，可以帮助我们理解该程序在这一种情况下的性能，但这并不是完全使人信服的证据。毕竟，如果使用其他的数据调用同一个程序，可能所花费的时间是完全不同的。简而言之，用特定的输入来测量函数所花费的时间类似于用特定的例子来测试函数。正如测试可能会暴露程序的错误，测量运行时间可能会得出非常规的、有关特定输入的执行性能，而并不保证能够得出一般情况下的程序性能。

本章介绍一种工具，它可以一般性地描述程序计算的时间。第一节引入这种工具，用几个例子来说明它，尽管这都是在一个非正式的基础之上的；第二节提供一个严格的定义；最后一节使用这种工具引入另一种 Scheme 的数据类型以及它的一些基本操作。

29.1 具体的时间和抽象的时间

先来研究我们非常熟悉的一个函数——*how-many*——的性能：

```
(define (how-many a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (how-many (rest a-list)) 1)]))
```

它读入一个表，计算该表包含了多少个元素。

下面是一个计算的例子：

```
  (how-many (list 'a 'b 'c))
= (+ (how-many (list 'b 'c)) 1)
= (+ (+ (how-many (list 'c)) 1) 1)
= (+ (+ (+ (how-many empty) 1) 1) 1)
= 3
```

其中只包含自然递归的步骤。其他步骤总是类似的，例如，要从原调用走到第一处自然递归，必须通过如下的步骤：

```
  (how-many (list 'a 'b 'c))

=(cond
   [(empty? (list 'a 'b 'c)) 0]
```

```
    [else (+ (how-many (rest (list 'a 'b 'c))) 1)])

=(cond
    [false 0]
    [else (+ (how-many (rest (list 'a 'b 'c))) 1)])

=(cond
    [else (+ (how-many (rest (list 'a 'b 'c))) 1)])
= (+ (how-many (rest (list 'a 'b 'c))) 1)
```

自然递归步骤之间唯一的不同就是使用了不同的 *a-list*。

如果把 *how-many* 作用于一个更短的表上，就只需要更少的自然递归步骤：

```
  (how-many (list 'e))
= (+ (how-many empty) 1)
=1
```

如果把 *how-many* 作用于一个更长的表上，就需要更多的自然递归步骤。在自然递归之间的步骤数目总是相同的。

这个例子说明，计算步骤的数目取决于输入的长度。这一点并不令人惊奇。可是，它还暗示，自然递归的数量是计算序列长度一个更好的度量。毕竟，我们可以由这个度量值以及函数的定义推导出真正的计算步骤数。因为这个原因，程序员们把某个程序输入的大小与计算过程中递归步骤数的关系称为该程序的抽象运行时间[1]。

在第一个例子中，输入的长度就是表的长度。更具体地说，如果表只包含一个元素，计算过程就只需要一次自然递归；对于包含两个元素的表，需要两次自然递归；对于包含 N 个元素的表，计算过程需要 N 个自然递归步骤。

并不是所有的函数都有这样一个统一的抽象运行时间的测度。观察我们遇到的第一个递归函数：

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [else (cond
           [(symbol=? (first a-list-of-symbols) 'doll) true]
           [else (contains-doll? (rest a-list-of-symbols))])]))
```

如果计算

```
(contains-doll? (list 'doll 'robot 'ball 'game-boy 'pokemon))
```

该调用并不需要自然递归的步骤。反之，对于表达式

```
(contains-doll? (list 'robot 'ball 'game-boy 'pokemon 'doll))
```

来说，计算过程所需的递归步骤与表的长度一样。换句话说，在最好的情况下，该函数可以马上找到答案；在最差的情况下，函数必须搜索整个输入表。

程序员不能假定输入总是最好的；他们必然希望输入不是可能最差的。取而代之的是，他们必须分析函数平均花费了多少时间。例如，*contains-doll?*可能——按平均数计算——在表的中部某处找到**'doll**。这样就可以说，如果输入包含 N 个元素，*contains-doll?*的抽象运行时间大约是：

$$\frac{N}{2}$$

[1] 因为这种度量方法忽略了具体每个基本步骤所使用的时间，以及整个计算过程所使用的时间，所以我们称之为抽象运行时间。

也就是说，它自然递归的次数是输入的一半。因为已经用了一种抽象的方法来度量函数的运行时间，因此可以忽略除数 2。更准确地说，我们假设每一个基本的步骤花费 K 个单元的时间。如果转而使用 $K/2$ 作这个常数，则可以认为：

$$K \cdot \frac{N}{2} = \frac{K}{2} \cdot N$$

这表明我们可以忽略常数因子。要表示省略了这样的常数，我们可以说，*contains-doll?*使用“阶为 N 的步骤数”从一个 N 个元素的表中寻找'doll。

现在，考虑图 12.1 中的标准排序函数。下面是该函数的一个针对较短输入的手工计算：

```
  (sort (list 3 1 2))
= (insert 3 (sort (list 1 2)))
= (insert 3 (insert 1 (sort (list 2))))
= (insert 3 (insert 1 (insert 2 (sort empty))))
= (insert 3 (insert 1 (insert 2 empty)))
= (insert 3 (insert 1 (list 2)))
= (insert 3 (cons 2 (insert 1 empty)))
= (insert 3 (list 2 1))
= (insert 3 (list 2 1))
= (list 3 2 1)
```

这个计算过程要比 *how-many* 或者 *contains-doll?*的计算过程复杂，虽然它也是由两个阶段组成的。在第一个阶段中，*sort* 的自然递归建立起多个对 *insert* 的调用，其数量与表的长度一样。在第二个阶段中，每一个 *insert* 调用分别处理长度为 1、2、3……的表，直到最后一个 *insert* 处理长度为原表长（减一）的表。

插入一个元素与寻找一个元素很类似，所以，并不令人吃惊的，*insert* 的操作类似于 *contains-doll?*。更具体地说，把 *insert* 作用于一个长度为 N 的表，可能会引起 N 次自然递归，也可能一次也没有。平均说来，我们认为它需要 $N/2$ 次递归，也就是说其阶为 N。因为总共有 N 次对 *insert* 的调用，因此总共平均自然递归调用 *insert* 的阶为 N^2。

总的说来，如果 l 包含 N 个元素，计算(*sort* l)需要 N 次 *sort* 的自然递归，阶为 N^2 次 *insert* 的自然递归。加起来共用了

$$N^2 + N$$

个步骤，但是，在习题 29.3.1 中，我们会看到，与之等价地，我们说插入排序需要的步骤数的阶为 N^2。

最后一个例子是函数 *max*：

```
;;max : ne-list-of-numbers -> number
;;求出一个非空数表中最大的数
(define (max alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (cond
            [(> (max (rest alon)) (first alon)) (max (rest alon))]
            [else (first alon)])]))
```

习题 18.1.12 研究了它的性能，还研究了一个观察上与它等价的、使用 local 的函数的性能。这里我们来研究它的抽象运行时间，而不是仅仅观测某些具体计算过程的运行时间。

从一个小例子开始：(*max* (list 0 1 2 3))，我们知道其结果是 3。下面是手工计算的第一个重要步骤：

```
(max (list 0 1 2 3))
```

```
= (cond
    [(> (max (list 1 2 3)) 0) (max (list 1 2 3))]
    [else 0])
```

这里必须计算左边的带下划线的自然递归。因为其值是 3，所以条件是 true，由此必须再计算第二个带下划线的自然递归。

接着来关注自然递归，其手工计算由类似的步骤开始：

```
  (max (list 1 2 3))
= (cond
    [(> (max (list 2 3)) 1) (max (list 2 3))]
    [else 1])
```

(*max* (list 2 3))又必须被计算两次，因为它生成了最大值。最终，即使是求(*max* (list 2 3))的最大值也需要两次自然递归：

```
  (max (list 2 3))
= (cond
    [(> (max (list 3)) 2) (max (list 3))]
    [else 2])
```

作为总结，对于 *max* 的每一次调用都需要两次自然递归：

计算表达式	需要计算两次
(*max* (list 0 1 2 3))	(*max* (list 1 2 3))
(*max* (list 1 2 3))	(*max* (list 2 3))
(*max* (list 2 3))	(*max* (list 3))

因此，手工计算一个长度为 4 的表的最大值，共需要进行八次自然递归。如果把 4（或者是一个更大的数）添加到表的尾部，则又需要两倍多的自然递归。因此，通常对于一个长为 N 的数表，如果最后一个数是最大的时候，计算 *max* 需要阶为

$$2^N$$

次的递归[1]。

这里考虑的是可能的情况中最差的，对 *max* 的抽象运行时间的分析解释了我们在习题 18.1.12 中所看到的现象，还解释了为什么 *max* 的变体——使用 local 表达式表示自然递归的返回值——运行起来更快：

```
;;max2 : ne-list-of-numbers -> number
;;求一个非空数表中最大的数
(define (max2 alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (local ((define max-of-rest (max2 (rest alon))))
            (cond
              [(> max-of-rest (first alon)) max-of-rest]
```

[1] 更精确地说，这个计算由 2^{N-1} 个步骤组成，但是

$$2^{N-1}=\frac{1}{2}\cdot 2^N,$$

这表明在忽略一个常数之后，阶为 2^N。

```
            [else (first alon)])]))))
```

这个函数不再重复计算表的其余部分的最大值，它只是两次访问变量的值，而这个变量就代表了表的其余部分的最大值。

习题

习题 29.1.1 数树是一个数或是一对数树。开发函数 *sum-tree*，求出一棵树中所有数的和。应该怎样度量树的大小？这个函数的抽象运行时间是什么？

习题 29.1.2 使用类似于我们计算(*max* (list 0 1 2 3))的方法，手工计算(*max2* (list 0 1 2 3))。它的抽象运行时间是什么？

29.2 "阶"的定义

现在介绍术语"阶"的严格描述，并解释为什么忽略某些常数是可行的。任何一个认真的程序员都必须十分熟悉这个概念。对于分析和比较程序的性能来说，它是最基本的方法。本章提供了这种思想的一个初步认识；关于计算技术的高级课程通常会提供一些更深入的描述。

在给出"阶"的定义之前，先来考虑一个关于阶的断言的具体例子。回忆一下，函数 F 可能需要阶为 N 的步骤，而函数 G 可能需要阶为 N^2 的步骤，尽管这两个函数计算的是同样的输入，得到的是同样的结果。现在假设 F 的基本时间常数是 **1000**，而 G 的基本时间常数是 **1**。一种比较两种阶的断言的方法是把抽象运行时间列成表格：

N	1	10	50	100	500	1000
$F(1000.N)$	1000	10000	50000	100000	500000	1000000
$G(N \cdot N)$	1	100	2500	10000	250000	1000000

初看起来，似乎这个表格说明 G 的性能要优于 F，因为对于同样长度的输入（N），G 的运行时间总是要少于 F。但是，更仔细的观察揭示了，当输入变大的时候，G 的优势在变小。事实上，对于长度为 1000 的输入，两个函数所需的步骤数是一样的，而从那以后，G 就总是比 F 慢。图 29.1 比较了两个表达式的图形。它显示了，对于某些有限大的数，线性曲线 $1000 \cdot N$ 位于曲线 $N \cdot N$ 的上方，但是从某一特定的数开始，就在下方了。

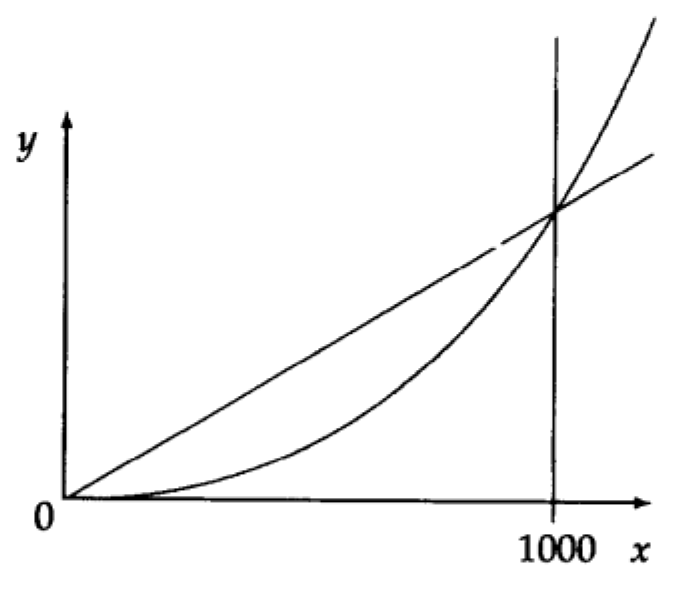


图 29.1 两个运行时间表达式的比较

这个具体的例子让我们回想起关于抽象运行时间的非正式讨论中的两个重要事实。第一，我们的抽

象描述总是两个量之间的关系：输入长度和求值过程中的自然递归数量之间的关系。更精确地说，该关系是一个数学上的函数，把输入长度的抽象度量映射到运行时间的抽象度量。第二，在我们比较函数的“阶”的性质的时候，例如

$$N, N^2, \text{或} \ 2^N$$

真正的意思是比较相应的函数（函数的参数是 N），并得出上述形式的结果。简而言之，关于阶的描述就是比较两个自然数（N）的函数。

比较两个 N 的函数是困难的，因为自然数有无穷多个。如果对于所有的自然数，函数 f 的值都比另一个函数 g *的值*要大，那么显然 f 大于 g。但是，如果对于少数的输入，这种大小关系不能成立，怎样判断两个函数的大小？比如说图 29.1 中的那个例子，对于 1000 个数，大小关系不成立，这时该怎么办？因为我们只是要得出近似的判断，程序员们和科学家们采用了一个数学概念，比较大于某个数的函数值，而忽略一些有限大的数。

阶（大 O）：给定一个自然数的函数 g，O（g）（读为“大 Og”）是一类自然数的函数，如果存在数 c 和足够大的数 bigEnough，使得对于所有的 $n \geqslant bigEnough$，

$$f(n) \le c \cdot g(n) \cdot$$

都成立，那么函数 f 在 $O(g)$ 之中。

回过头考虑前述的 F 和 G 的性能。对于前者来说，我们假定它花费的时间取决于函数

$$f(N) = 1000 \cdot N;$$

后者的性能服从函数 g：

$$g(N) = N^2$$

使用大 O 的定义，我们可以说，f 在 O（g）中，因为，对于所有的 $n \geq 1000$，

$$f(n) \le 1 \cdot g(n),$$

这表示 $bigEnough = 1000$，$c = 1$。

更重要的是，大 O 的定义提供了一种描述关于函数运行时间的简略表达方法。例如，今后可以说 *how-many* 的运行时间是 O（N）。记住，N 是数学上的函数 g（N）$=N$ 标准的简写。类似地，我们说，在最差的情况下，*sort* 的运行时间是 O（N^2），*max* 的运行时间是 O（2^N）。

最后，大 O 的定义解释了我们为什么在比较抽象运行时间时不必注意比率常数。考虑 *max* 和 *max2*。我们知道 *max* 在最差情况下的运行时间在 O（2^N）中，*max2* 则在 O（N）中。比方说，我们需要求 10 个数的表的最大值。假设 *max* 和 *max2* 的每个基本步骤所使用的时间是大致相等的，*max* 需要 $2^{10}=1024$ 步，而 *max2* 只需要 10 步，这表示 *max2* 运行得更快。现在，即使 *max2* 的基本步骤需要花去的时间是 *max* 的基本步骤的两倍，*max2* 仍然要快大约 50 倍。另外，如果我们把输入表的长度加倍，*max* 表面上的缺点就完全消失了。一般来说，输入越长，比率常数就越不重要。

习题

习题 29.2.1 在本章的第一节中，我们曾说函数 f（n）$=n^2+n$ 属于 O（n^2）类。给定一组数 c 和 *bigEnough*，证实这个断言。

习题 29.2.2 考虑函数 f（n）$=2^n$ 和 g（n）$=1000 \cdot n$。证明 g 属于 O（f），这表示理论上说，f 比 g 更（或者至少一样）耗费时间。如果输入的长度必定在 3 和 12 之间，那么哪个函数更好？

习题 29.2.3 比较 f（n）$=n \log n$ 和 g（n）$=n^2$。f 是否属于 O（g），g 是否属于 O（f）？

29.3 向量初探

到现在为止，我们还没有探讨过从结构体或者表中提取数据需要多少时间。既然有了一种描述一般判断的工具，那么就让我们仔细地研究一下这个基本的计算步骤。回忆一下本书前一部分中的最后一个问题：在一张图中寻找路线。程序 *find-route* 需要两个辅助函数：*find-route/list* 以及 *neighbors* 。事实上，开发 *neighbors* 只是一道练习题（参见习题 28.1.2），因为从表中查找某个值是一种常见的程序任务。

neighbors 一种可能的定义是：

```
;; neighbors : node graph -> (listof node)
;; 在 graph 中查找 node
(define (neighbors node graph)
  (cond
    [(empty? graph) (error 'neighbors "can't happen")]
    [else (cond
            [(symbol=? (first (first graph)) node) (second (first graph))]
            [else (neighbors node (rest graph))])]))
```

这个函数类似于 *contains-doll?*，并且性能也大约相同。更具体地说，如果我们假设 *graph* 是包含 *N* 个节点的表，那么 *neighbors* 就属于 $O(N)$ 。

考虑到在执行 *find-route* 的过程中，每一个阶段都要使用到 *neighbors*，*neighbors* 可能是一个瓶颈。事实上，如果我们要寻找的路线包括了 *N* 个节点（最大值），*neighbors* 就会被调用 *N* 次，所以算法在 *neighbors* 中就需要 $O(N^2)$ 个步骤。

与表不同，从结构体中提取值是一种常数时间的操作。初一看，这一事实似乎暗示我们使用结构体来表示图。但是，仔细地观察说明这种想法并不能顺利地运作。如果我们能够操作节点的名称，使用名称存取节点的邻居，图的算法就能很好地工作。节点的名称可以是一个符号，也可能是该节点在图中的标号。一般而言，我们真正希望程序设计语言提供的是
一种复合值的数据类型，长度可变，基于“关键字”查找只需要常数时间。

因为这个问题非常普遍，Scheme 以及许多其他的语言都提供至少一种内建的解决方法。

这里，我们学习向量类型。向量是一种定义明确的数学数据类型，有其特有的基本操作。就我们的用途而言，懂得如何创建它们，如何提取值，以及如何辨认它们就足够了：

1. 操作 vector 类似于 list，它的参数是任意数量的值，用它们建立一个复合值：一个向量。例如，(vector *V-0* ... *V-n*)创建一个从 *V-0* 到 *V-n* 的向量。

2. DrScheme 还提供了一个类似于 build-list 的向量操作，它被称作 build-vector。它是这样工作的：

```
(build-vector N f) = (vector (f 0) ... (f (- N 1)))
```

也就是说，build-vector 以自然数 *N* 和作用于自然数的函数 *f* 为参数。它分别把 *f* 作用于 0，……，*N-1*，从而创建一个包含 *N* 个元素的向量。

3. 操作 vector-ref 从向量中提取出一个值，只使用常数时间，也就是说，对于在 0 和 n（包括）之间的 *i*：

```
(vector-ref (vector V-0 ... V-n) i) = V-i
```

简而言之，从向量中提取一个值所需的时间是 $O(1)$ 。

如果传给 vector-ref 一个向量和一个自然数，而该自然数小于 0 或者大于 *n*，vector-ref 会产生一个错误信号。

4. 操作 vector-length 给出某个向量所包含的元素数目：

```
(vector-length (vector V-0 ... V-n)) = (+ n 1)
```

5．操作 vector?判定向量：

```
(vector? (vector V-0 ... V-n)) = true
(vector? U) = false
```

如果 U 不是由 **vector** 创建的值。

我们可以把向量当作有限的小范围内的自然数的函数。向量的元素的取值范围是所有 Scheme 值的类型。我们也可以把向量当作表格，它把（有限小范围内的）自然数与 Scheme 值联系起来。如果用数表示节点的名称，则可以用向量来表示图 28.1 和图 28.3 中的那些图。例如：

A	B	C	D	E	F	G
0	1	2	3	4	5	6

使用这种变换，可以用向量表示图 28.1 中的图：

```
(define Graph-as-list          (define Graph-as-vector
 '((A (B E))                     (vector (list 1 4)
   (B (E F))                             (list 4 5)
   (C (D))                               (list 3)
   (D ())                                empty
   (E (C F))                             (list 2 5)
   (F (D G))                             (list 3 6)
   (G ())))                              empty))
```

左边的定义是原来的基于表的表示法；右边的定义是基于向量的表示法。向量的第 i 个字段就是第 i 个节点的邻居的表。

node 和 *graph* 的数据定义相应改变。我们假设 N 是给定的表中节点的数目：

> *node*（节点）是在 0 和 N-1 之间的自然数。

> *Graph*（图）是 *node* 的向量：(vectorof (listof *node*))。

记号(vectorof *X*)类似于(listof *X*)，表示包含某种未指定的数据类型 *X* 的向量。

现在，我们可以重新定义 *neighbors*：

```
;; neighbors : node graph -> (listof node)
;;查找 graph 中的 node
(define (neighbors node graph)
  (vector-ref graph node))
```

这样，查找一个节点的邻居就是常数时间的操作了，并且在研究 *find-route* 的抽象运行时间时，我们可以忽略这个开销。

习题

习题 29.3.1　测试新的 *neighbors* 函数。使用第 17.8 节中的策略，用布尔值表达式表示测试。

习题 29.3.2　修改 *find-route* 程序的其余部分，使之适用于新的向量表示法。修改习题 28.1.3 至习题 28.1.5 中的测试，检验新的程序。

分别用两个 *find-route* 程序计算在图 28.1 中的图中从节点 A 到节点 E 的路线，并且测量它们所使用的时间。回忆一下，**(time** *expr***)**测量求 *expr* 的值所花费的时间。在测量时间的时候，最好多次计算 *expr*，比如说 1000 次。这样可以得到更精确的测量值。

习题 29.3.3　把图 28.3 中的循环图转变为图的向量表示法。

在真正可以使用向量编程之前，必须理解其数据定义。这种情况相当于我们第一次遇到表。我们知道 vector 和 cons 一样，是 Scheme 提供的，但是目前并没有可以用来指导程序设计的数据定义。

所以，让我们来仔细观察一下向量。粗略地说，vector 类似于 cons：基本操作 cons 构造表，基本操作 vector 构造向量。使用表来编程通常意味着使用选择器 first 和 rest 来编程，而使用向量来编程必然意味着使用 vector-ref 来编程。不过，与 first 和 rest 不同，vector-ref 需要向量以及它的一个下标，这表示使用向量编程真正要考虑的是下标，而下标是自然数。

先看一些简单的例子，从而巩固这个抽象的判断。下面是第一个例子：

```
;; vector-sum-for-3 : (vector number number number) -> number
(define (vector-sum-for-3 v)
  (+ (vector-ref v 0)
     (vector-ref v 1)
     (vector-ref v 2)))
```

函数 *vector-sum-for-3* 读入三个数组成的向量，求出它们的总和。它使用 vector-ref 来提取出三个数，并把它们加起来。在三个选择器表达式中，下标在变化，而向量保持不变。

考虑第二个更有趣的例子：*vector-sum*，即 *vector-sum-for-3* 的一般化。它读入任意长的数向量，求出数的总和：

```
;;vector-sum : (vectorof number) -> number
;;计算 v 中数的总和
(define (vector-sum v) ...)
```

下面是一些例子：

```
(= (vector-sum (vector -1 3/4 1/4))
   0)
(= (vector-sum (vector .1 .1 .1 .1 .1 .1 .1 .1 .1 .1))
   1)
(= (vector-sum (vector))
   0)
```

最后一个例子表明，即使向量是空的，我们也需要一个合理的答案。就像 empty 一样，在这种情况下我们使用 **0** 来当答案。

问题是，一个与 *v* 相关的自然数，即它的长度，并不是 *vector-sum* 的参数。当然，*v* 的长度只是指出 *v* 中有多少个元素需要被处理，而这反过来涉及 *v* 的合法下标数。这迫使我们开发一个辅助函数，读入一个向量和一个自然数：

```
;; vector-sum-aux : (vectorof number) N -> number
;; 计算 v 中相对于 i 的数的总和
(define (vector-sum-aux v i) ...)
```

自然，我们选择 *v* 的长度作 *i* 的初始值，这表明完成后的 *vector-sum* 是这样的：

```
(define (vector-sum v)
  (vector-sum-aux v (vector-length v)))
```

在这个定义的基础上，还可以把 *vector-sum* 的例子修改成 *vector-sum-aux* 的例子：

```
(= (vector-sum-aux (vector -1 3/4 1/4) 3)
   0)
(= (vector-sum-aux (vector .1 .1 .1 .1 .1 .1 .1 .1 .1 .1) 10)
   1)
(= (vector-sum-aux (vector) 0)
   0)
```

不幸的是，这并没有阐明第二个参数的功能。想知道它的功能，我们需要进入设计过程的下一个阶段：设计模板。

在设计两个参数的函数的模板的时候，必须先决定哪个参数必须被处理，也就是说，在计算的过程中，哪一个参数会变化。*vector-sum-for-3* 的例子说明，在这种情况下是第二个参数。因为这个参数属于自然数类型。使用有关自然数的设计诀窍：

```
(define (vector-sum-aux v i)
  (cond
    [(zero? i) ...]
    [else ... (vector-sum-aux v (sub1 i)) ...]))
```

虽然我们最初把 *i* 看作向量的长度,但是模板说明我们应该把它当作 *vector-sum-aux* 必须处理的元素的数目，从而就是 *v* 的一个下标。

通过详细地描述关于 *i* 的使用，自然地得出了 *vector-sum-aux* 一个更好的用途说明：

```
;; vector-sum-aux : (vectorof number) N -> number
;; 计算 v 中下标为[0, i) 的数的总和
(define (vector-sum-aux v i)
  (cond
    [(zero? i) ...]
    [else ... (vector-sum-aux v (sub1 i)) ...]))
```

把 *i* 排除在外是很自然的，因为 *i* 的初始值是(vector-length *v*)，而这不是一个（合法的）下标。

要把模板转化为完整的函数定义，我们分别考虑 cond 的每一个子句：

1. 如果 *i* 是 **0**，那么没有元素需要被处理，因为在 **0** 和 *i* 之间（不包括 *i*）没有向量的字段。因此返回值是 **0**。

2. 否则，(*vector-sum-aux v* (sub1 *i*))计算出 *v* 中在 0 和(sub1 *i*) [不包括]之间的数的和。这就留下了下标为(sub1 *i*)的向量字段没有处理，而按照用途说明，它必须被包括进来。加上(vector-ref *v* (sub1 *i*))就可得到返回值：

```
(+ (vector-ref v (sub1 i)) (vector-sum-aux v (sub1 i)))
```

完整的程序请参见图 29.2。

如果手工计算 *vector-sum-aux* 的一个例子，我们就会发现，随着 *i* 逐步减小，函数按照从右向左的顺序提取出向量中的数。一个自然的问题是：我们能不能把这个顺序倒过来。换一种说法：是不是存在一个函数，按照从左向右的顺序提取数？

为了回答这个问题，我们再开发一个函数，从第一个可行的下标—0 开始，处理位于(vector-length *v*)之下的自然数。开发这样的函数只是第 11.4 节中介绍的自然数变量设计诀窍的另一个例子。图 29.3 给出了新的函数。新辅助函数的参数现在以 0 为初始值，逐渐增加到(vector-length *v*)。对

```
(lr-vector-sum (vector 0 1 2 3))
```

的手工计算显示了 *vector-sum-aux* 确实是从左向右提取 *v* 的元素的。

```
;; vector-sum : (vectorof number) -> number
;; 计算 v 中数的总和
(define (vector-sum v)
  (vector-sum-aux v (vector-length v)))
;; vector-sum-aux : (vectorof number) N -> number
;; 计算 v 中下标为[0, i]的数的总和
(define (vector-sum-aux v i)
  (cond
    [(zero? i) 0]
    [else (+ (vector-ref v (sub1 i))
             (vector-sum-aux v (sub1 i)))]))
```

图 29.2　计算向量中数的总和（第一版）

```
;; lr-vector-sum : (vectorof number) -> number
;; 计算 v 中数的总和
(define (lr-vector-sum v)
  (vector-sum-aux v 0))

;; vector-sum : (vectorof number) -> number
;; 计算 v 中下标为[i, (vector-length v))的数的总和
(define (vector-sum-aux v i)
  (cond
    [(= i (vector-length v)) 0]
    [else (+ (vector-ref v i) (vector-sum-aux v (add1 i)))]))
```

图 29.3　计算向量中数的总和（第二版）

lr-vector-sum 的定义说明了我们为什么需要学习另一种自然数类型的定义。有时候，我们必须要倒计数到 0，但是在其他的情况下，从 0 计数到某个自然数同样的重要，并且更为自然。

这两个函数还表明对区间的考虑是多么的重要。辅助的向量处理函数处理给定向量的区间。一个良好的用途说明精确地描述了函数处理的区间。事实上，一旦我们准确地理解了区间的说明，要给出完整的函数相对来说是简单的。在本章的最后一节，回过头再学习向量处理函数时，我们就会明白这一点的重要性。

习题

习题 29.3.4　手工计算(*vector-sum-aux* (vector -1 3/4 1/4) 3)，只需给出主要的步骤。使用 DrScheme 的单步执行检查该手工计算。这个函数是用哪种顺序相加向量中的数的？

使用 local 表达式定义一个单一的函数 *vector-sum*，然后去除内部函数定义中的向量参数。为什么可以这样做？

习题 29.3.5　手工计算(*lr-vector-sum* (vector -1 3/4 1/4))，只需给出主要的步骤。使用 DrScheme 的单步执行检查该手工计算。这个函数是用哪种顺序相加向量中的数的？

使用 local 表达式定义一个单一的函数 *lr-vector-sum*，然后去除内部函数定义中的向量参数。另外，引入那些在计算过程中经常需要求值的表达式的定义。这样做有什么好处？

习题 29.3.6　与 *vector-sum* 对应的、基于表的函数是 *list-sum*：

```
;;list-sum : (listof number) -> number
;;计算 alon 中数的总和
(define (list-sum alon)
```

```
  (list-sum-aux alon (length alon)))
;; list-sum-aux : N (listof number) -> number
;; 计算 alon 中前 L 个数的总和
(define (list-sum-aux L alon)
  (cond
    [(zero? L) 0]
    [else (+ (list-ref alon (sub1 L)) (list-sum-aux (sub1 L) alon))]))
```

这个程序的开发中并没有使用表的结构定义，而是使用了表的长度——一个自然数——作为设计过程的指导元素。

其结果是，该定义使用 Scheme 的 *list-ref* 函数来访问表中的每一个元素。用 list-ref 在长为 N 的表中查找一个元素是一个 *O*（*N*）的操作。确定 *sum*（定义于第 9.5 节）、*vector-sum-aux* 和 *list-sum-aux* 的抽象运行时间。关于程序设计，这说明了什么？

习题 29.3.7 开发函数 *norm*，该函数读入一个数向量，求出其中数的平方和的平方根。*norm* 的另一个名称是 *distance-to-0*，因为如果把向量理解为一个点，这个函数就返回从该点到原点的距离。

习题 29.3.8 开发函数 *vector-contains-doll?*，该函数读入一个符号向量，判断该向量是否包含符号'doll。如果包含，它返回'doll 字段的下标；否则，返回 false。

确定 *vector-contains-doll?*的抽象运行时间，并把它与 *contains-doll?*的抽象运行时间比较（我们在前一节中讨论过 *contains-doll?*的抽象运行时间了）。

现在来考虑如下的问题。假设我们要表示一个符号的集合，对于这个集合，我们所关心的唯一问题就是判断它是否包含某个给定的符号。该集合最好使用哪一种数据表示法：表还是向量？为什么？

习题 29.3.9 开发函数 *binary-contains?*，该函数读入一个有序的数向量和一个关键字，其中的关键字也是一个数。函数的目标是：如果关键字存在于向量中，求出它的下标，否则返回 false。使用第 27.3 节中介绍的二分查找算法。

确定 *binary-contains*?的抽象运行时间，并把它与 *contains*?的抽象运行时间比较，*contains*?函数以线性的方式（*vector-contains-doll?*使用的方式）在向量中查找关键字。

假设我们要表示一个数的集合，对于这个集合，我们所关心的唯一问题就是判断它是否包含某个给定的数。该集合最好使用哪一种数据表示法：表还是向量？为什么？

习题 29.3.10 开发函数 *vector-count*，该函数读入符号向量 *v* 和符号 *s*，返回 *v* 中 *s* 出现的次数。

确定 *vector-count* 的抽象运行时间，并把它与 *count* 的抽象运行时间比较，这里的 *count* 函数计算在一个符号表中 s 出现了多少次。

假设要表示一个符号的集合，对于这个集合，我们所关心的唯一问题就是它多少次包含了某个给定的数。该集合最好使用哪一种数据表示法：表还是向量？为什么？习题 29.3.8、习题 29.3.9 以及本习题说明了什么？

访问向量中的元素是一类编程问题，而构造向量是一种与之完全不同的问题。如果知道向量中元素的数目，我们可以用 vector 构造它。不过，如果想要编写出处理一大类向量的函数，独立于它们的长度，就需要使用 build-vector。

考虑如下的简单例子：假设我们用一个向量表示物体的速度。例如，(vector 1 2)表示一个物体在平面上的运动速度，单位时间内，它向右移动一个单元，向下移动两个单元。作为对比，(vector -1 2 1)是一个物体在空间中的运动速度：在 6 个时间单位中，该物体在 *x* 方向上移动了-6 个单位，在 *y* 方向上移动了 12 个单位，在 *z* 方向上移动了 6 个单位。我们把(vector -6 12 6)称为该物体在 6 个时间单位中的位移。

开发一个函数，计算某个速度为 *v* 的物体在 *t* 个时间单位内的位移：

```
;; displacement : (vectorof number) number -> (vectorof number)
```

```
;; 计算 v 和 t 所对应的位移
(define (displacement v t) ...)
```

对于某些例子来说，计算位移相当简单：

```
(equal? (displacement (vector 1 2) 3)
        (vector 3 6))
(equal? (displacement (vector -1 2 1) 6)
        (vector -6 12 6))
(equal? (displacement (vector -1 -2) 2)
        (vector -2 -4))
```

只需用一个数去乘向量的每一个成分，从而生成一个新的向量。

对于程序的问题来说，例子的意义是说明 *displacement* 必须创建一个和 *v* 一样长的表，而且必须用 *v* 中的元素来计算出新的向量。可以这样构造一个和 *v* 一样长的向量：

```
(build-vector (vector-length v) ...)
```

现在需要把…替换成一个函数，计算新向量的第 0 个元素、第 1 个元素等等：

```
;; new-item : N -> number
;; 计算新向量在第 i 个位置中的内容
(define (new-item index) ...)
```

根据讨论，只需用 *t* 去乘(vector-ref *v i*)就可以了。

来看一看完整的定义：

```
;; displacement : (vectorof number) number -> (vectorof number)
;; 计算 v 和 t 所对应的位移
(define (displacement v t)
  (local ((define (new-item i) (* (vector-ref v i) t)))
    (build-vector (vector-length v) new-item)))
```

这里局部定义的函数不是递归的，因而我们可以用一个简单的 lambda 表达式替换它：

```
;; displacement : (vectorof number) number -> (vectorof number)
;; 计算 v 和 t 所对应的位移
(define (displacement v t)
  (build-vector (vector-length v) (lambda (i) (* (vector-ref v i) t))))
```

数学家们把这个函数称为内积。其他向量运算，在 Scheme 中，也可以用自然的方式加以设计。

习题

习题 29.3.11 开发函数 *id-vector*，该函数读入一个自然数，生成一个向量，包含那么多个 1。

习题 29.3.12 开发函数 *vector+*和 *vector-*，计算两个向量的逐个元素之和及逐个元素之差。更确切地说，这两个函数都读入两个向量，进行相应的处理后生成一个向量。假定给定的两个向量是等长的。另外，开发函数 *checked-vector+*和 *checked-vector-*（检查给定的两个向量是否等长）。

习题 29.3.13 开发函数 *distance*，该函数读入两个向量，计算它们之间的距离。把两个向量之间的距离看作为它们之间的直线的长度。

习题 29.3.14 开发一种大小为 $n \times n$ 的棋盘的表示法，其中的 n 属于 **N**。然后开发如下两个关于棋盘的函数：

```
;;build-board : N (N N -> boolean) -> board
;;创建一个棋盘，大小为 n×n。
```

```
;;用(f i j)填充下标为 i 和 j 的位置
(define (build-board n f) ...)

;; board-ref : board N N -> boolean
;; 访问 a-board 中下标为 i 和 j 的位置
(define (board-ref a-board i j) ...)
```

现在可以使用向量而不是表来运行第 28.2 节中的程序了吗？检查习题 28.2.3 和习题 28.2.4 的解答。

习题 29.3.15 矩阵是由数组成的棋盘。使用习题 29.3.14 中的棋盘表示法来表示矩阵

$$\begin{vmatrix} 1 & 0 & -1 \\ 2 & 0 & 9 \\ 1 & 1 & 1 \end{vmatrix}$$

使用 *build-board* 开发函数 *transpose*，该函数创建原矩阵的转置矩阵，也就是矩阵按从左上角到右下角的对角线的镜像。例如，给定的矩阵被转变为

$$\begin{vmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \\ -1 & 9 & 1 \end{vmatrix}$$

更为一般地说，在（*i*，*j*）的元素变成了在（*j*，*i*）的元素。





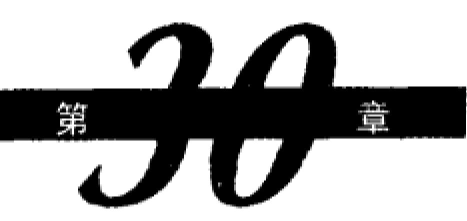
第六部分

知识累积





知识的丢失



在设计递归函数的时候，我们并不考虑它们使用的背景，即我们并不关心它是第 1 次被调用，还是第 100 次被递归调用。函数会按照它们的用途说明来工作，这就是在设计函数主体时需要知道的全部东西。

虽然这种背景无关原则极大地方便了函数设计，但它有时也会导致问题。本章通过两个例子来说明递归计算过程中所出现的知识丢失现象。本章第一节简述知识丢失现象如何使一个结构递归函数变得复杂，效率变得低效；第二节说明丢失的知识可能在算法中造成致命的错误。

30.1 一个与结构处理相关的问题

假设我们知道有一序列的点位于同一直线上，并且知道了（从起点开始）相邻两点之间的距离，要求出从起点到每个点的距离。例如已知如下图所示的直线：

```
•    50    •   40   •      70      •   30   •   30   •
```

图中的数表示相邻两点之间的距离，要做的是根据该图，求出从最左的点到每一个点的距离：

```
0          50        90                160     190     220
•          •         •                 •       •       •
```

```
;; relative-2-absolute : (listof number) -> (listof number)
;; 把相对距离的表转换成绝对距离的表
;; 表中的第一个元素表示到起点的距离
(define (relative-2-absolute alon)
  (cond
    [(empty? alon) empty]
    [else (cons (first alon)
            (add-to-each (first alon) (relative-2-absolute (rest alon))))]))

;; add-to-each : number (listof number) -> (listof number)
;; 把 n 加到 alon 中的每个数上
(define (add-to-each n alon)
  (cond
    [(empty? alon) empty]
    [else (cons (+ (first alon) n) (add-to-each n (rest alon)))]))
```

图 30.1　把相对距离转换成绝对距离

设计一个执行该功能的函数只是一个结构程序设计的练习。图 30.1 给出了完成后的 Scheme 程序。当给定的表不是 empty 时，自然递归会计算出(rest *alon*)中其余的点到表的第一个元素的距离。因为递归调用产生的结果表中第一个元素并不是真正的起点，而是与起点有一定的距离，所以必须把(first *alon*)加至表的每一个元素。该操作，即把一个数加到表的每一个元素上，需要使用一个辅助函数。

虽然开发这样一个函数相当简单，但是把它作用于越来越大的表时，问题出现了。考虑计算如下的定义：[1]

(define *x* (*relative-2-absolute* (list 0 ... *N*)))

当 *N* 增大时，计算所需的时间飞速增长：[2]

N	计算所需的时间
100	220
200	880
300	2050
400	5090
500	7410
600	10420
700	14070
800	18530

当表的长度从 100 增为 200 时，所需的时间翻了 4 倍。从 200 增为 400、从 300 增为 600，这个关系也近似成立。

习题

习题 30.1.1 使用 map 和 lambda 重新给出 *add-to-each*。

习题 30.1.2 给出 *relative-2-absolute* 理论上的运行时间。

提示：手工计算表达式

```
(relative-2-absolute (list 0 ... N))
```

把 *N* 替换成 1，2，3 等，问每一次分别需要多少次递归调用 *relative-2-absolute* 和 *add-to-each*？

考虑简化了的问题，即这两个函数执行的“工作量”，其结果是相当惊人的。假如通过手工计算来转换距离表，我们只需沿着直线逐一将绝对距离加上相邻两点的距离值。

我们重新来设计这个函数，使其工作方式更接近于手工计算时使用的方法。新的函数仍然是一个表处理函数，所以我们从适当的模板开始：

```
(define (rel-2-abs alon)
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (rel-2-abs (rest alon)) ...]))
```

现在设想(*rel-2-abs* (list 3 2 7))的“计算过程”：

```
  (rel-2-abs (list 3 2 7))
= (cons ... 3 ...
(convert (list 2 7)))
```

[1] 构建这个表最方便的方法是计算(**build-list (add1 *N*) identity**)。

[2] 使用不同的计算机，计算所需的时间各不相等。这里的数据采集自一台使用 Linux 操作系统的奔腾 166 计算机。测量运行时间也相当困难。至少，每一个计算都必须反复执行多次，而最终的数据应当是多次测量的平均值。

```
= (cons ... 3 ...
    (cons ... 2 ...
      (convert (list 7))))
= (cons ... 3 ...
    (cons ... 2 ...
      (cons ... 7 ...
        (convert empty))))
```

返回表的第一个元素显然应该是3，要构造这样一个表毫无困难。但是，表的第二个元素应该是(+ 3 2)。然而 *rel-2-abs* 的第二个实例并没有任何方法可以“获知”原表的第一个元素是3，这个“知识”被丢失了。

换一种说法，问题的关键所在是，递归函数与上下文独立。函数处理表(**cons** *N L*)中 *L* 的方法与其处理表(**cons** *K L*)中 *L* 的方法完全一样。事实上，如果我们（不是通过递归，而是直接）把表 *L* 传给这个递归函数，它还是一样处理这个表。结构递归函数的这种特性使其容易设计，但也正是这种特性有时造成了函数的结构比必需的更复杂，并且这种复杂性会影响到函数的性能。

为了补偿这种丢失的“知识”，我们给递归函数增加一个额外的参数：*accu-dist*。在这个把相对距离转换成绝对距离的例子中，这个新的参数代表了累积的距离，即一个累积器。它的初值必然是0。随着函数的运行，对表中数的不断处理，就把所有的数加到累积器上。

下面就是修改后函数的定义：

```
(define (rel-2-abs alon accu-dist)
  (cond
    [(empty? alon) empty]
    [else (cons (+ (first alon) accu-dist)
                (rel-2-abs (rest alon) (+ (first alon) accu-dist)))]))
```

这时，递归调用使用表的其余部分以及新的从当前点到起点的距离作参数。虽然这意味着这两个参数同时在改变中，但是第二个参数的改变严格地依赖于第一个参数，所以说该函数还是一个普通的表处理函数。

使用例子来运行 *rel-2-abs*，其过程显示了使用累积器能够极大地简化计算过程：

```
  (rel-2-abs (list 3 2 7) 0)
= (cons 3 (rel-2-abs (list 2 7) 3))
= (cons 3 (cons 5 (rel-2-abs (list 7) 5)))
= (cons 3 (cons 5 (cons 12 (rel-2-abs empty 12))))
= (cons 3 (cons 5 (cons 12 empty)))
```

输入表的每一个元素都只被处理一次。当 *rel-2-abs* 到达参数表的末端时，其结果已完成确定，不再需要进一步计算。一般来说，函数作用于一个长度为 *N* 的表只需要进行 *N* 次自然递归。

新定义的函数还有一个小问题：它需要两个参数，所以它并不与 *relative-2-absolute* 完全等价。更糟糕的是，有人可能意外地把 *rel-2-abs* 错误地作用于一个表和一个不等于0的数上。使用图30.2中的函数，把 *rel-2-abs* 包含在一个局部定义中，就可以解决这两个问题。现在，*relative-2-absolute* 和 *relative-2-absolute2* 之间（对于使用者来说）没有差别了。

```
;; relative-2-absolute2 : (listof number) -> (listof number)
;; 把相对距离的表转换成绝对距离的表
;; 表中的第一个元素表示到起点的距离
(define (relative-2-absolute2 alon)
  (local ((define (rel-2-abs alon accu-dist)
            (cond
              [(empty? alon) empty]
              [else (cons (+ (first alon) accu-dist)
                      (rel-2-abs (rest alon) (+ (first alon) accu-dist)))])))
    (rel-2-abs alon 0)))
```

图 30.2 使用累积器把相对距离转换成绝对距离

30.2 一个关于生成递归的问题

再次考虑第 28 章路径寻找问题：给定若干个节点以及它们之间的连接，求出从 *orig* 节点到 *dest* 节点的路径。这里只考虑简单图问题的简化版本，从其中每个节点出发到另一个节点有且只有一个单向连接。

观察图 30.3，该图有从 A 到 F 的 6 个节点和 6 个连接。要想从 A 走到 E，可以经过 B、C 到达 E。我们可以从 A 走到 E，但是无法从 F 走到 A（或者是任何一个 F 以外的节点）。

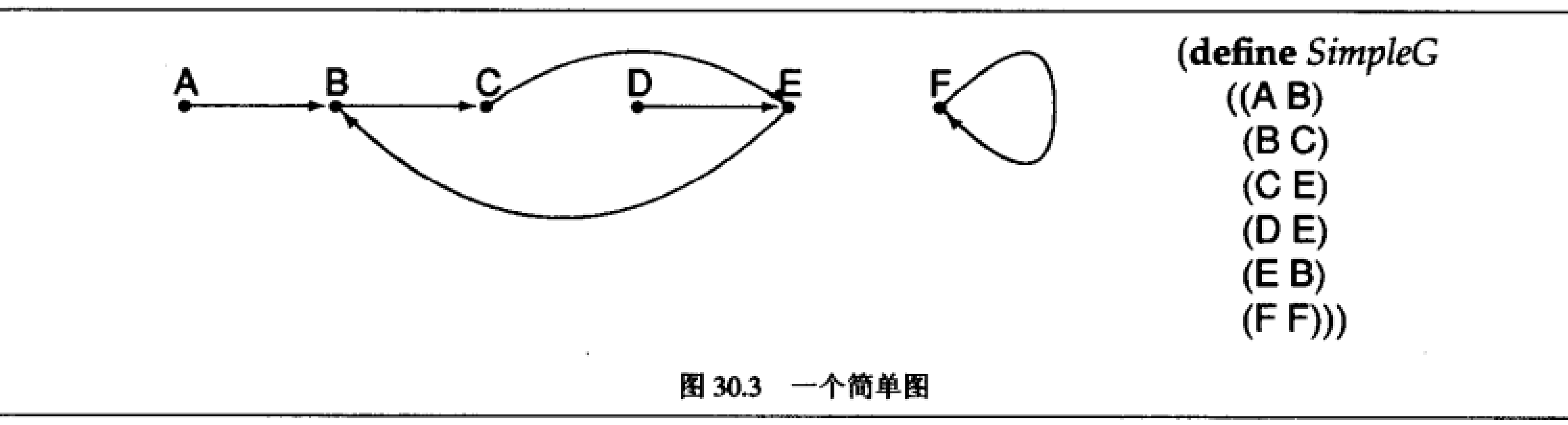


图 30.3 一个简单图

图 30.3 的下方给出了该图的 Scheme 定义，节点用两个符号组成的表表示，第一个符号是该节点的标号，第二个符号表示从该节点出发的连接所到达的节点。与此相关的数据定义是：

> *node*（节点）是符号。

> *pair* 是由两个 *node* 组成的表：
> (cons *S* (cons *T* empty))
> 其中 *S* 和 *T* 是符号。

> *simple-graph*（简单图）是由 *pair* 组成的表：
> (listof *pair*)

在图中寻找路径是一个生成递归问题。有了数据定义，有了例子，现在可以给出函数头部：

```
;; route-exists? : node node simple-graph -> boolean
;; 判断在 sg 中是否存在一条从 orig 到 dest 的路径
(define (route-exists? orig dest sg) ...)
```

现在所需要的是生成递归设计诀窍的四个基本问题的答案：

什么是平凡可解的问题？如果节点 *orig* 和 *dest* 是同一个节点，该问题就是平凡的。

对应的解是什么？太简单了，是 true。

如何生成新的问题？如果节点 *orig* 和 *dest* 不是同一个节点，那么所能做的事只有一件，即从 *orig* 节点走到下一个节点，然后判断是否存在从它到 *dest* 的路径。

如何把解联系起来？当找到新问题的解后，无需做任何事情，如果 *orig* 的后一个节点能够到达 *dest*，那么 *orig* 也能。

接着，只需把这些问题的答案用 Scheme 表示出来，就可以得到所需的算法。图 30.4 给出了完整的函数，包括在简单图中查找下一个节点的函数。

```
;; route-exists? :  node node simple-graph -> boolean
;; 判断在 sg 中是否存在一条从 orig 到 dest 的路径
(define (route-exists? orig dest sg)
  (cond
    [(symbol=? orig dest) true]
    [else (route-exists? (neighbor orig sg) dest sg)]))

;; neighbor :  node simple-graph -> node
;; 求出在 sg 中 a-node 可到达的下一个节点
(define (neighbor a-node sg)
  (cond
    [(empty? sg) (error "neighbor:  impossible")]
    [else (cond
            [(symbol=? (first (first sg)) a-node)
             (second (first sg))]
            [else (neighbor a-node (rest sg))])]))
```

图 30.4　在简单图中寻找路径（第一个版本）

即使是不经意地观察一下这个函数，也会发现一个大问题。如果图中不存在一条从 *orig* 到 *dest* 的路径，那么该函数应该返回 false。但是，这个函数中根本没有 false 存在。反过来，我们要问，如果图中不存在从 *orig* 到 *dest* 的路径，那么这个函数干了些什么事？

让我们再来观察图 30.3，在这个简单图中，并不存在从 C 到 D 的路径。所以让我们看看如果把'C、'D 和 *SimpleG* 传给 *route-exists?*，它会如何运行：

```
  (route-exists? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
= (route-exists? 'E 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
= (route-exists? 'B 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
= (route-exists? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)))
```

手工计算显示，在该函数递归过程中，它反复调用自己，换句话说，计算过程永远也不会终止。

与上一节中的 *relative-2-absolute* 类似，这个问题仍然是由“知识”丢失造成的。和 *relative-2-absolute* 一样，*route-exists?*是依照设计递归函数的标准诀窍开发的，与背景独立，也就是说，它并不“知道”自己在递归链中是第一次被调用，还是第一百次被调用。对于 *route-exists?*这个例子来说，这意味着该函数并不“知道”当前调用是否与以前的调用完全一样。

解决这个问题的方法与上一节给出的方法类似。我们增加累积器 *accu-seen*，以表的形式表示以前遇

到过的出发点。显然，它的初始值是 **empty**。每次函数调用时，都会把 *orig* 放入 *accu-seen* 中，然后再移动到 *orig* 的下一个节点。

下面是修改后的 *route-exists?*，我们把它命名为 *route-exists-accu?*：

```
;; route-exists-accu? : node node simple-graph (listof node) -> boolean
;; 判断在 sg 中是否存在一条从 orig 到 dest 的路径
;; 假设在 accu-seen 中的节点已经被检查过了
;; 并且从这些点出发都不能找到解
(define (route-exists-accu? orig dest sg accu-seen)
  (cond
    [(symbol=? orig dest) true]
    [else (route-exists-accu? (neighbor orig sg) dest sg
                              (cons orig accu-seen))]))
```

仅仅加上一个新的参数并没有解决问题，但是，正如下面的手工计算显示的，它提供了解决问题的基础：

```
  (route-exists-accu? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F)) empty)
= (route-exists-accu? 'E 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '(C))
= (route-exists-accu? 'B 'D '((A B) (B C) (C E) (D E) (E B) (F F)) '(E C))
= (route-exists-accu? 'C 'D '((A B) (B C) (C E) (D E) (E B) (F F))
                      '    (B E C))
```

不同于第一个版本的函数，修改后的函数不会再次用完全相同的参数调用自己。虽然在第三次调用时，前三个参数与第一次调用是一样的，但是累积器与第一次调用不同。第一次调用时，该参数的值是 empty，而现在它是'(B E C)。这表示在寻找从'C 到'D 的路径的过程中，节点'B、'E 和'C 已经被作为出发点检查过了。

现在我们所要做的就是在函数中使用累积的知识。对于这个问题，我们要做的就是检查 *orig* 是否已经存在于 *accu-seen* 中了。如果 *accu-seen* 已经包含 *orig*，那么问题的解就应当是 false。图 30.5 给出了 *route-exists?* 的第二个版本，即 *route-exists2?*。在 *route-exists2?*的定义中使用了 *contains*，即我们遇到的第一个递归函数（参见本书的第二部分），它判断某个符号是否在某个表中出现。

```
;; route-exists2? : node node simple-graph -> boolean
;; 判断在 sg 中是否存在一条从 orig 到 dest 的路径
(define (route-exists2? orig dest sg)
  (local ((define (re-accu? orig dest sg accu-seen)
            (cond
              [(symbol=? orig dest) true]
              [(contains orig accu-seen) false]
              [else (re-accu? (neighbor orig sg) dest sg (cons orig accu-seen))])))
    (re-accu? orig dest sg empty)))
```

图 30.5 在简单图中寻找路径（第二个版本）

*route-exists2?*的定义同时也解决了两个次要的问题。通过使用 local 表达式定义累积函数，确保了第一次调用 *re-accu?*函数时，传给 *accu-seen* 的初始值总是 empty。*route-exists2?*的合约以及用途说明与 *route-exists?*完全一致。

*route-exists2?*和 *relative-to-absolute2* 有一点不同：*relative-to-absolute2* 与原先的函数在功能上等价，而 *route-exists2?*是对 *route-exists?*函数的彻底改进。毕竟，对于某些输入，*route-exists?*根本不能运行，而 *route-exists2?*改正了这个致命的错误。

习题

习题 30.2.1　完成图 30.5 中的定义，然后使用简单图来测试它。使用第 17.8 节中给出的策略把测试结果表达为布尔表达式。手工检查该函数，说明它能够对参数'A、'C 和 *SimpleG* 输出正确的结果。

习题 30.2.2　编辑图 30.5 中的函数，使得用 local 定义的那个函数只读入在计算过程中改变的参数。

习题 30.2.3　开发一个基于向量的简单图表示法，然后修改图 30.5 中的函数，让它使用基于向量的简单图表示法。

习题 30.2.4　修改图 28.2 中 *find-route* 和 *find-route/list* 的定义，使得它们即使在两次遇到同一个起始节点时也输出 false。



设计带累积器的函数



第 30 章通过两个例子显示了递归函数需要记住附加的知识。有时候，使用累积器能使函数易于理解；有时候，为了使函数能够正确工作，还不得不使用累积器。无论是哪一种情况，我们总是先选择一种可用的设计诀窍，检查完成的函数，然后再修改它。换一种更一般的说法，添加一个累积器，即一个记住知识的参数，是在完成了函数设计之后而不是在设计函数之前进行的工作。

设计一个带累积器的函数的关键是：

1. 认识到这个函数受益于使用累积器，或者函数需要使用累积器；
2. 理解累积器代表什么东西。

本章前两节讨论这两个问题，其中第二个问题比较难，第三节通过例子说明怎样精确地定义累积器。更具体地说，这一章通过使用带累积器的辅助函数对几个标准递归函数进行修正。

31.1 认识累积器的必要性

要认识累积器的必要性并不是一件简单的事。我们已经看到了两种需要使用累积器的原因，它们是添加累积器最主要的理由。无论是哪种原因，关键的步骤都是要先建立一个完整的、基于设计诀窍的函数。然后，研究这个函数，寻找下列特征中的一个：

1. 如果这个函数结构上是递归的，并且递归调用的结果被交给一个辅助函数处理，那么应当考虑使用一个累积器。

用 *invert* 函数当例子：

```
;; invert : (listof X) -> (listof X)
;; 构造 alox 的倒转
;; 结构递归
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item (first alox) (invert (rest alox)))]))

;; make-last-item : X (listof X) -> (listof X)
;; 把 an-x 加到 alox 的尾部
;; 结构递归
(define (make-last-item an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (make-last-item an-x (rest alox)))]))
```

递归调用的结果产生了表的其余部分的倒转。通过调用 *make-last-item*，表的第一个元素被添加到表

的其余部分倒转的尾部，从而产生了整个表的倒转。其中第二个函数，即辅助函数也是递归的。因此，我们确定这个函数可能需要使用累积器。现在，如第 30.1 节所做的，就是通过研究一些手工计算的例子，观察是否需要一个累积器。

2. 如果正在处理的函数是一个生成递归函数，我们就面临着一个相当困难的任务。我们的目的必然是理解算法是否不能从输入产生所要的输出。要是那样的话，添加一个参数可能会对算法有所帮助。因为这种情况相当复杂，我们把对它的讨论推迟到 32.2 节。

这两种情况最普遍，但绝不是仅有的，为了加强理解，在后面的章节，我们会讨论另一些可能性。

31.2 带累积器的函数

如果认定需要使用带累积器的函数，分两步引入：

设立累积器：首先，我们必须理解累积器需要记住的知识是什么，以及怎样记住它。例如，在把相对距离转化成绝对距离的过程中，把目前所遇到的总距离累积起来就足够了。对于寻找路径问题，我们需要用累积器记住迄今为止已检查过的节点，因此，前一个累积器是数，而后一个累积器是节点的表。

讨论累积过程最好的方法是，通过 local 定义引入一个带累积器函数的模板，并重命名（原来的）函数的参数，使之与辅助函数的参数不同。

我们来观察 *invert* 的例子：

```
;; invert : (listof X) -> (listof X)
;; 构造 alox 的倒转
(define (invert alox0)
  (local (;; accumulator ...
      (define (rev alox accumulator)
        (cond
          [(empty? alox) ...]
          [else
          ... (rev (rest alox) ... ( first alox) ... accumulator)
          ... ])))
    (rev alox0 ...)))
```

这样就得到了 *invert* 的模板定义，其中包含了一个辅助函数 *rev*。这个辅助函数的参数要比 *invert* 多出一个：（累积器 *accumulator*）。递归调用中的方框指出我们需要一个表达式，用来维持累积过程，并且这个过程依赖于 *accumulator* 和(first *alox*)的当前值，*rev* 的值将要被遗忘。

显然，*invert* 不能忘记任何东西，因为它所做的只是倒转表中的元素，因此可以要求把 *rev* 所遇到的所有元素都记住。这意味着：

1. *accumulator* 代表了一个表。
2. *accumulator* 代表了 *alox0* 中所有在 *rev* 的参数 *alox* 之前的元素。

从分析的第二部分来看，关键的问题是要能够区分原来的参数 *alox0* 以及现在的参数 *alox*。

知道了累积器的大致用途，接着来考虑它的第一个值应当是什么，而对应的递归又干些什么。在 local 表达式的主体中，调用 *rev* 时，它接收到的参数是 *alox0*，这表示它还没有遇到任何的元素。*accumulator* 的初始值是 empty。再次调用 *rev* 时，它正好遇到了一个附加的元素：(first *alox*)。为了记住这一点，我们可以把它用 cons 连接到累积器的当前值上。

下面是改进后的定义：

```
;; invert : (listof X) -> (listof X)
```

```
;; 构造 alox 的倒转
(define (invert alox0)
  (local (;; accumulator 是 alox0 中 alox 之前的元素的倒转表
      (define (rev alox accumulator)
        (cond
          [(empty? alox) ...]
          [else
          ... (rev (rest alox) (cons (first alox) accumulator))
          ...])))
    (rev alox0 empty)))
```

观察揭示 *accumulator* 并不只是 *alox0* 前部的元素，而是它们反过来排列的表。

使用累积器：一旦决定了使用累积器累积什么知识，如何记住知识，下面转而讨论如何在函数中使用它。

对于例子 *invert*，答案几乎是显然的。如果 *accumulator* 是 *alox0* 中 *alox* 之前所有元素的倒转，那么，如果 *alox* 是空的，*accumulator* 就代表了 *alox* 的倒转。换一种说法，如果 *alox* 是 empty，*rev* 的答案就是 *accumulator*，而这就是在两种情况下所需的答案：

```
;; invert : (listof X) -> (listof X)
;; 构造 alox 的倒转
(define (invert alox0)
  (local (;;accumulator 是 alox0 中 alox 之前的元素的倒转表
      (define (rev alox accumulator)
        (cond
          [(empty? alox) accumulator]
          [else
          (rev (rest alox) (cons (first alox) accumulator))])))
    (rev alox0 empty)))
```

这个开发过程的关键是要精确地描述 *accumulator* 的角色。一般来说，累积器不变式描述了函数的参数、当前辅助函数的参数以及带累积器的函数运行期间必须维持的累积器三者之间的关系。

31.3 把函数转换成带累积器的变体

在设计诀窍中，最复杂的部分就是给出累积器不变式。没有累积器不变式就无法给出带累积器的函数。给出累积器不变式显然是一件需要大量练习的技巧性工作，所以这一节使用三个小规模的、易于理解的、不需要累积的结构函数进行练习。本节的最后部分是一系列习题。

第一个例子，考虑函数 *sum*：

```
;; sum : (listof number) -> number
;; 计算 alon 中数的总和
;; 结构递归
(define (sum alon)
  (cond
    [(empty? alon) 0]
    [else (+ (first alon) (sum (rest alon)))]))
```

下面是把它改成带累积器的函数的第一步：

```
;; sum : (listof number) -> number
;; 计算 alon0 中数的总和
```

```
(define (sum alon0)
  (local (;; accumulator ...
      (define (sum-a alon accumulator)
        (cond
          [(empty? alon) ...]
          [else
         ... (sum-a (rest alon) ... ( first alon) ... accumulator)
         ... ])))
    (sum-a alon0 ...)))
```

正如第一步所示，我们用一个 local 来定义 *sum-a* 的模板，增加了一个累积器，并且重命名了 *sum* 的参数。

我们的目的是开发 *sum* 带累积器的变体，要做到这一点，必须考虑 *sum* 是怎样工作的，以及其目的是什么。与 *rev* 类似，*sum-a* 一个一个地处理表中的数，其目的是把这些数加起来。这表明 *accumulator* 代表目前为止遇到的数的和：

```
...
 (local (;; accumulator是alon0中在alon之前的数的总和
      (define (sum-a alon accumulator)
        (cond
          [(empty? alon) ...]
          [else
         ... (sum-a (rest alon) (+ (first alon) accumulator))
         ... ])))
    (sum-a alon0 0)))
```

如果调用 *sum-a*，必须把 **0** 传给 *accumulator*，因为目前我们还没有处理过 *alon* 中的任何数。对于第二个子句，必须把(first *alon*)加到 *accumulator* 之上，使不变式对函数调用保持不变。

得出了精确的不变式，剩下的工作就简单了。如果 *alon* 是 empty，*sum-a* 返回 *accumulator*，因为它就代表了当前 *alon* 中所有数的和。图 31.1 给出了带累积器的 *sum* 的最终定义。

```
;; sum : (listof number) -> number
;; 计算 alon0 中数的总和
(define (sum alon0)
  (local (;;accumulator 是 alon0 中在 alon 之前的数的总和
          (define (sum-a alon accumulator)
            (cond
              [(empty? alon) accumulator]
              [else (sum-a (rest alon) (+ (first alon) accumulator))])))
(sum-a alon0 0)))

;; ! : N -> N
;; 计算 n · (n - 1) · ... · 2 · 1
(define (! n0)
  (local (;; accumulator 是[n0, n]中所有自然数的乘积
          (define (!-a n accumulator)
            (cond
              [(zero? n) accumulator]
              [else (!-a (sub1 n) (* n accumulator))])))
    (!-a n0 1)))
```

图 31.1　一些简单的带累积器的函数

我们用相同的输入来比较 *sum* 原先的定义和带累积器的定义各自产生的结果：

```
  (sum (list 10.23 4.50 5.27))
= (+ 10.23 (sum (list 4.50 5.27)))
= (+ 10.23 (+ 4.50 (sum (list 5.27))))
= (+ 10.23 (+ 4.50 (+ 5.27 (sum empty))))
= (+ 10.23 (+ 4.50 (+ 5.27 0)))
= (+ 10.23 (+ 4.50 5.27))
= (+ 10.23 9.77)
= 20.0
```

```
  (sum (list 10.23 4.50 5.27))
= (sum-a (list 10.23 4.50 5.27) 0)
= (sum-a (list 4.50 5.27) 10.23)
= (sum-a (list 5.27) 14.73)
= (sum-a empty 20.0)
= 20.0
```

在左边的表中可以看到普通的递归函数是怎样把数表一步一步拆开的，每一步都放上一个加法操作。在右边的表中可以看到带累积器的函数是怎样随着每一步运行把数累加起来的。另外，我们还可以看到，每一次调用 *sum-a*，不变式都保持不变。当最后 *sum-a* 被作用于 empty 时，累积器就是最终的结果，由 *sum-a* 返回。

习题

习题 31.3.1　上述两个函数的第二个不同之处是加法的顺序。*sum* 原先的定义是从右加到左，而带累积器的函数是从左加到右。对于精确数来说，这一点不同对于最终产生的结果没有影响。但是对于非精确数来说，差别是巨大的。

考虑如下的定义：

```
(define (g-series n)
  (cond
    [(zero? n) empty]
    [else (cons (expt -0.99 n) (g-series (sub1 n)))]))
```

把 *g-series* 作用于一个自然数，会产生一个递减的几何级数的开始部分（即前个 *n* 元素）（参见第 23.1 节）。

使用不同的函数求表中诸元素之和，会得到差别较大的结果。分别用原来的 *sum* 和带累积器的 *sum* 计算表达式：

```
(sum (g-series #i1000))
```

然后再计算

```
(* 10e15 (sum (g-series #i1000)))
```

结果表明，取决于不同的背景，两者之间的差别可以是任意大。

作为第二个例子，让我们讨论本书第二部分中的阶乘函数：

```
;; ! : N -> N
;; 计算 n · (n - 1) · ... · 2 · 1
;; 结构递归
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n)))]))
```

relative-2-absolute 和 *invert* 处理的是表，而阶乘函数处理的是自然数，它的模板是处理 **N** 的函数模

板。按照惯例我们从创建!的 **local** 定义开始：

```
;; ! : N -> N
;; 计算n · (n - 1) · .... · 2 · 1
(define (! n0)
  (local (;; accumulator ...
      (define (!-a n accumulator)
        (cond
          [(zero? n) ...]
          [else
          ... (!-a (sub1 n) ... n ... accumulator) ...])))
    (!-a n0 ...)))
```

这个程序草稿说明，如果把!作用于自然数 n，那么*!-a* 先处理 n，然后处理 n-1、n-2、……，直到 **0** 为止。因为我们的目的是把这些数乘起来，所以累积器应该是所有*!-a* 遇到过的数的乘积：

```
...
  (local (;; accumulator 是在 n0（包括）
          ;; 和 n（不包括）之间的所有数的乘积
         (define (!-a n accumulator)
         (cond
          [(zero? n) ...]
          [else
         ... (!-a (sub1 n) (* n accumulator)) ...])))
    (!-a n0 1)))
```

要使不变式在一开始就保持正确，必须使累积器的初值为 **1**。在*!-a* 递归的过程中，必须把累积器的当前值乘以 *n*，重新建立不变式。

从*!-a* 的用途说明可以看到，如果 *n* 是 **0**，那么累积器是 *n*，……，**1** 的乘积，它就是所需的答案。所以，像 *sum* 一样，在第一种情况下，*!-a* 返回 *accumulator*，而在第二种情况下，进行递归调用。图 31.1 给出了完整的定义。

手工计算并比较两个版本函数是很有启发意义的：

```
  (! 3)
= (* 3 (! 2))
= (* 3 (* 2 (! 1)))
= (* 3 (* 2 (* 1 (! 0))))
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6
```

```
  (! 3)
= (!-a 3 1)
= (!-a 2 3)
= (!-a 1 6)
= (!-a 0 6)
= 6
```

左边一栏中给出了原先函数的工作过程，右边一栏中给出了带累积器函数的工作过程。这两个函数都遍历自然数直到 0 为止，但是原来的函数只是确定乘法的运算过程，而新的函数在运行中真正地把数乘了起来。另外，右边一栏还显示了新函数是怎样维持累积器不变式的。每一次调用时，累积器都是 **3** 至 *n* 的乘积，其中 *n* 是*!-a* 的第二个参数。

习题

习题 31.3.2　与 *sum* 类似，*!*反过来执行基本的计算步骤（乘法）。令人惊讶的是，这一点负面影响了函数的执行。请使用 DrScheme 的 time 程序来检测这两个函数 1000 次计算(*!* 20)分别需要多少时间。

提示：（1）设计函数：

```
;; many: N (N -> N) -> true
;; 计算 n 次(f 20)
(define (many n f) ...)
```

（2）计算(**time** an-expression)可以测定求 an-expression 用了多少时间。

最后研究一个处理简化二叉树的函数，这个例子显示了带累积器的函数并不仅限于处理单一自我引用的数据定义。事实上，除了表和自然数，它还常常被用来处理各种复杂的数据定义。

下面是简化二叉树的结构定义：

```
(define-struct node (left right))
```

相应的数据定义为：

> *binary-tree*（二叉树，简称树）是下列二者之一：
> 1. Empty。
> 2. (make-node *tl tr*)，其中 *tl* 和 *tr* 都是树。

这些树不包含任何信息，都是以 empty 结尾，图 31.2 给出了许多不同的树，从中可以看出，可以把树看成一个图，即把 empty 看成一个普通的点，而把 make-node 看成一个结合两棵树的点。

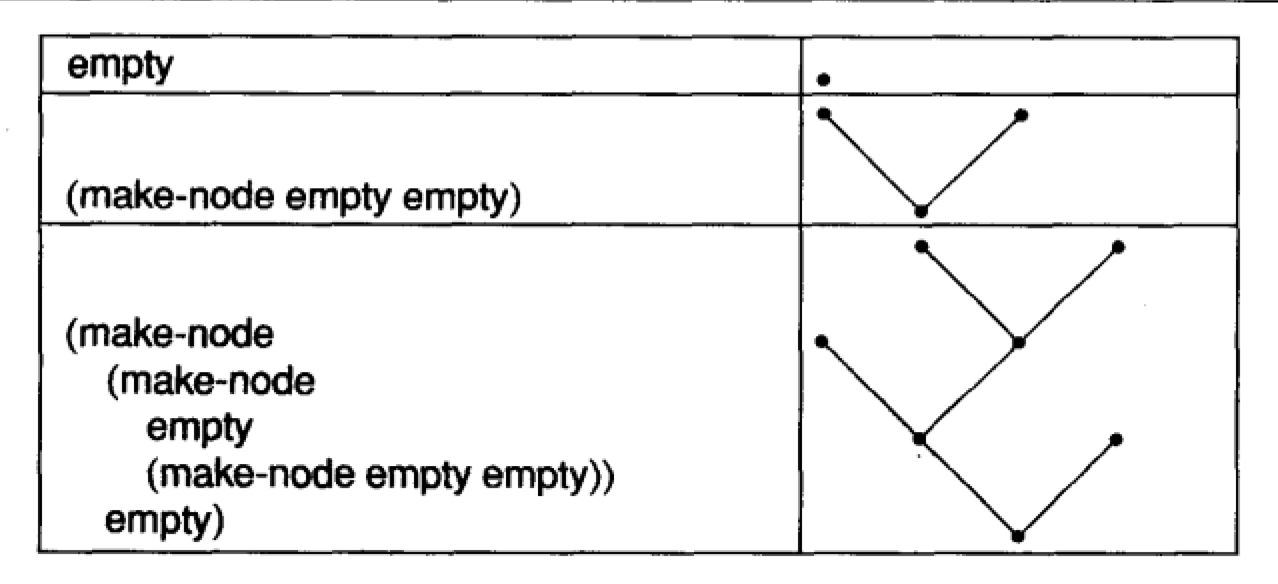


图 31.2　一些朴素二叉树

通过二叉树的图形表达，我们可以轻易地看出树的属性。例如，可以数出它包含多少个节点，有多少个 empty，或者它有多高。让我们来观察函数 *height*，它读入树并测定该树有多高：

```
;; height : tree -> number
;; 测量 abt0 的高度
;; 结构递归
(define (height abt)
  (cond
    [(empty? abt) 0]
    [else (+ (max (height (node-left abt)) (height (node-right abt))) 1)]))
```

跟数据定义一致，函数定义中有两处自我引用。要把这个函数转化成带累积器的函数，可按照标准的方法进行。首先，把合适的模板放入 local 定义：

```
;; height : tree -> number
;; 测量 abt0 的高度
(define (height abt0)
  (local (;; accumulator ...
      (define (height-a abt accumulator)
        (cond
          [(empty? abt) ...]
          [else
         ... (height-a (node-left abt)
                       ... ( node-right abt) ... accumulator) ...
         ... (height-a (node-right abt)
                       ... ( node-left abt) ... accumulator) ... ]))))
    (height abt0 ...)))
```

按照惯例，现在的问题是要确定累积器应当表示什么知识。

一个明显的选择是令 *accumulator* 为一个数，表示 *height-a* 已经处理过的 *node* 的数目。一开始，它经历过 0 个节点；随着树的分解、节点的处理，累积器的值不断递增：

```
...
  (local (;; accumulator 表示 height-a 在
          ;; 从 abt0 到 abt 的路上遇到过的节点数
      (define (height-a abt accumulator)
        (cond
          [(empty? abt) ...]
            [else
         ... (height-a (node-left abt)  (+ accumulator 1)) ...
         ... (height-a (node-right abt) (+ accumulator 1)) ...])))
    (height abt0 0))
```

更确切地说，累积器不变式是 *height-a* 走到树 *abt* 所用的步数。

对于基本情况来说，结果仍然是 *accumulator*，这是因为它代表了某一特定路径的长度。但是，与前两个例子不同，它还不是最终的结果。在第二个 **cond** 子句中，新的函数要处理两个高度。既然我们只对其中大的一个感兴趣，就用 Scheme 的 *max* 操作把它选出来。

```
;; height : tree -> number
;; 测量 abt0 的高度
(define (height abt0)
  (local (;; accumulator 表示 height-a 在
          ;; 从 abt0 到 abt 的路上遇到过的节点数
          (define (height-a abt accumulator)
            (cond
              [(empty? abt) accumulator]
              [else (max (height-a (node-left abt)  (+ accumulator 1))
                         (height-a (node-right abt) (+ accumulator 1)))])))
    (height-a abt0 0)))
```

图 31.3　带累积器的 height

图 31.3 给出了 *height* 完整的定义。我们所要进行的最后一个步骤是通过手工计算检验新的函数。这里使用图 31.2 中最复杂的例子进行检验：

```
  (height (make-node
            (make-node empty
                       (make-node empty empty))
           empty))

= (height-a (make-node
             (make-node empty
                 (make-node empty empty))
              empty)
0)

= (max (height-a
         (make-node empty
            (make-node empty empty))
        1)
         (height-a empty 1))

= (max (max
         (height-a empty 2)
         (height-a (make-node empty empty) 2))
(height-a empty 1))

= (max (max
           (height-a empty 2)
           (max (height-a empty 3) (height-a empty 3)))
(height-a empty 1))

= (max (max
           2
          (max 3 3))
        1)

= 3
```

这说明，在递归的每一步，height-a 增加累积器的值，而在路径顶端，累积器就代表了所经过的直线的数目。手工计算还显示了不同分支的结果在每一个分叉点被处理。

习题

习题 31.3.3 设计带累积器的函数 *product*，该函数计算一个数表内诸元素的乘积。解释累积器代表了什么。

习题 31.3.4 设计带累积器的函数 *how-many*，该函数给出一个表的元素数。解释累积器代表了什么。

习题 31.3.5 设计带累积器的函数 *add-to-pi*，不使用+，把一个自然数加到 *pi* 之上（参见第 11.5 节）。解释累积器代表了什么。一般化这个函数，使它不使用+而能求取两个数之和，其中第一个数是自然数。

习题 31.3.6 设计函数 *make-palindrome*，该函数读入一个非空表，围绕表中的最后一个元素倒写这个表，从而构造出一个回文。例如，把 *make-palindrome* 作用于单词“abc”，可以得到“abcba”。

习题 31.3.7　设计函数 *to10*，该函数读入一个数字表，产生对应的数，表中的第一个数是最高位。例如：

```
(= (to10 (list 1 0 2))
102)


(= (to10 (list 2 1))
   21)
```

一般化这个函数，使它以一个基数 b 以及一个 b 进制数位的表为参数，函数产生该表代表的十进制数的（以 10 为基数的）值。基数是一个在 2 和 10 之间的数。b 进制的数位是一个在 0 和 b－1 之间的数。

例如：

```
(= (to10-general 10 (list 1 0 2))
   102)
(= (to10-general 08 (list 1 0 2))
   66)
```

提示：在第一个例子中，由

$$1\cdot10^2+0\cdot10^1+2\cdot10^0=((1\cdot10+0)\cdot10)+2$$

得到答案；第二个例子由

$$1\cdot8^2+0\cdot8^1+2\cdot8^0=((1\cdot8)+0)\cdot8)+2$$

得到答案。

习题 31.3.8　设计函数 *is-prime?*，该函数读入一个自然数，如果这个自然数是素数，返回 true，反之，返回 false。数 *n* 是素数的条件是，它不能被 2 和 *n*－1 之间的任何一个数整除。

提示：由 **N**[>=*1*]的设计诀窍可得如下模板：

```
;; is-prime? : N[>=1] -> boolean
;; 判断 n 是不是一个素数
(define (is-prime? n)
  (cond
    [(= n 1) ...]
    [else ... (is-prime? (sub1 n)) ...]))
```

从这个框架中，我们可以立即得出结论：当这个函数递归时，它立即忘记了初始的参数 *n*。如果要判断 *n* 能否被 2……n－1 整除，*n* 必然是需要的，所以这表示我们要设计一个带累积器的局部函数，使得它在递归时能够记住 *n*。

缺陷：第一次遇到带累积器的函数的人常常产生这样的印象，带累积器的函数总是比与之对应的递归函数要易于理解，并且运行速度要快，这两点都是错误的。虽然不太可能全面地讨论这个问题，但我们可以来看一个小例子。考虑如下的表格：

一般的阶乘函数	带累积器的阶乘函数
5.760	5.970
5.780	5.940

5.800	5.980
5.820	5.970
5.870	6.690
5.806	6.110

这个表格就是习题 31.3.2 的答案。具体来说，左边的一栏给出了用纯阶乘函数计算 1000 次(! 20)的时间（以秒为单位）；右边的一栏给出了用带累积器的函数进行同样计算所花费的时间。最后一行是两栏的平均值。这个表格说明，带累积器的阶乘函数的性能往往要比原来的阶乘函数差。



使用累积器的更多例子

第 32 章

这一章给出了三个补充练习。这三个练习需要全面的技巧：遵照诀窍设计（包括生成递归的设计诀窍）以及使用各种用途的累积器。

32.1 补充练习：有关树的累积器

(**define-struct** child (father mother name date eyes))

> 家谱树节点（简写 *ftn*）是下列二者之一：
>
> 1. *empty*。
>
> 2. (**make-child** *f m na da ec*)，其中 *f* 和 *m* 是 *ftn*，*na* 和 *ec* 是符号，而 *da* 是数。

```
;; all-blue-eyed-ancestors : ftn -> (listof symbol)
;; 用 a-ftree 中所有的蓝眼睛祖先构造一个表
(define (all-blue-eyed-ancestors a-ftree)
  (cond
    [(empty? a-ftree) empty]
    [else (local ((define in-parents
                   (append (all-blue-eyed-ancestors (child-father a-ftree))
                           (all-blue-eyed-ancestors (child-mother a-ftree)))))
           (cond
             [(symbol=? (child-eyes a-ftree) 'blue)
              (cons (child-name a-ftree) in-parents)]
             [else in-parents]))]))
```

图 32.1 用 *blue-eyed-ancestor?* 搜集蓝眼睛成员

图 32.1 再次使用了 14.1 节中家谱树的结构体和数据定义。第 14.1 节还开发了函数 *blue-eyed-ancestor?*，判断祖先家谱树是否包含蓝眼睛成员。与之不同的是，图 32.1 中的 *all-blue-eyed-ancestors* 函数搜集某个给定的家谱树中所有蓝眼睛成员的名字。

这个函数的结构就是树处理函数，它包含两个子句，一句用来处理空白树，另一句用来处理 *child* 节点，后一个子句又包含两处自我引用：每个双亲各一个。这些递归调用分别搜集父亲以及母亲的家族树中蓝眼睛祖先的名字，再通过 append 把这两个表结合成一个。

append 函数是结构递归函数。这里，它处理 *all-blue-eyed-ancestors* 自然递归产生的两个表。按照第

17.1 节的论述，我们很自然地选中这个函数，将其转化为带累积器的函数，过程如下：

```
;; all-blue-eyed-ancestors : ftn -> (listof symbol)
;; 用 a-ftree 中所有的蓝眼睛祖先构造一个表
(define (all-blue-eyed-ancestors a-ftree0)
  (local (;; accumulator ...
          (define (all-a a-ftree accumulator)
            (cond
              [(empty? a-ftree) ...]
              [else
               (local ((define in-parents
                         (all-a ... (child-father a-ftree) ...
                       ... accumulator ...)
                         (all-a ... (child-mother a-ftree) ...
                       ... accumulator ...)))
                 (cond
                   [(symbol=? (child-eyes a-ftree) 'blue)
                    (cons (child-name a-ftree) in-parents)]
                   [else in-parents]))])))
    (all-a a-ftree0 ...)))
```

下一个目标是给出累积器不变式。累积器一般的用途是记住在 *all-a* 遍历树时丢失的、有关 *a-ftree0* 的知识。观察图 32.1 中的定义，我们找到了两处递归调用，一处处理(child-father *a-ftree*)，这意味着这个对 *all-blue-eyed-ancestors* 的调用丢失了有关 *a-ftree* 的母亲的知识；另一处处理 *a-ftree* 母亲，相反，无需任何有关父亲的知识。

这样就有两个选择：

1. 在累积器被传给父亲的树时，它应当表示到目前为止遇到的所有蓝眼睛的祖先，包括母亲的家谱树中的蓝眼睛祖先。

2. 另一个选择是让累积器代表树中丢失的元素，更确切地说，当 *all-a* 处理父亲的家谱树时，它用累积器记住母亲的树（以及所有其他它以前没有遇到的东西）。

下面分别研究这两种可能，从第一种开始。

一开始，*all-a* 并没有遇到过家谱树中的任何一个节点，所以 *accumulator* 是 empty。当 *all-a* 要遍历父亲的家谱树时，必须建立一个表，表示树中所有将要被遗忘的蓝眼睛的祖先，也就是母亲的树。这表示我们应给出如下的累积器不变式：

```
;; accumulator 是在 a-ftree0 中到 a-ftree 的路上遇到的
;; 母方树上蓝眼睛祖先的表
```

要维持不变式，必须收集母亲一方树上的祖先。既然 *all-a* 的目的是收集这些祖先，在将 *all-a* 作用于(child-father *a-ftree*)时，使用表达式

```
(all-a (child-mother a-ftree) accumulator)
```

来计算新的累积器。把所有这些东西放到一起，第二个 cond 子句就是：

```
(local ((define in-parents
          (all-a (child-father a-ftree)
          (all-a (child-mother a-ftree)
          accumulator))))
  (cond
    [(symbol=? (child-eyes a-ftree) 'blue)
```

```
    (cons (child-name a-ftree) in-parents)]
  [else in-parents]))
```

剩下来的就是给出第一个 cond 子句的答案。既然 *accumulator* 代表了目前所遇到的所有蓝眼睛的祖先，它就是所需的计算结果。图 32.2 是完整的定义。

```
;; all-blue-eyed-ancestors : ftn -> (listof symbol)
;; 用 a-ftree 中所有的蓝眼睛祖先构造一个表
(define (all-blue-eyed-ancestors a-ftree)
  (local (;; accumulator 是在 a-ftree0 中到 a-ftree 的路上遇到的
          ;; 母亲一方的树上的蓝眼睛祖先的表
          (define (all-a a-ftree accumulator)
            (cond
              [(empty? a-ftree) accumulator]
              [else
               (local ((define in-parents
                         (all-a (child-father a-ftree)
                                (all-a (child-mother a-ftree) accumulator))))
                 (cond
                   [(symbol=? (child-eyes a-ftree) 'blue)
                    (cons (child-name a-ftree) in-parents)]
                   [else in-parents]))])))
    (all-a a-ftree empty)))
```

图 32.2　搜集蓝眼睛的祖先（第一个带累积器的版本）

对于第二种选择，我们希望累积器表示还没有处理过的家谱树的表。因为累积器目的的不同，让我们称这个累积器参数为 *todo*：

;; *todo* 是遇到过但还没有处理过的家谱树的表

与带累积器的 *invert* 类似，*all-a* 赋给 *todo* 的初始值是 empty。通过扩充自然递归的表，*all-a* 更新 *todo* 的值：

```
(local ((define in-parents
          (all-a (child-father a-ftree)
            (cons (child-mother a-ftree) todo))))
  (cond
    [(symbol=? (child-eyes a-ftree) 'blue)
     (cons (child-name a-ftree) in-parents)]
    [else in-parents]))
```

现在的问题是，当 *all-a* 作用于 empty 树时，*todo* 并不代表结果，而是代表剩下来的要处理的东西。要处理所有这些家谱树，*all-a* 必须依次作用于它们。当然，如果 *todo* 是 empty，那么就没有剩余的东西，结果也就是 empty。如果 *todo* 是表，我们取出表中的第一个元素进行处理，然后是其余部分：

```
(cond
  [(empty? todo) empty]
  [else (all-a (first todo) (rest todo))])
```

表的其余部分就是现在剩下来的、要处理的东西了。

```
;; all-blue-eyed-ancestors : ftn -> (listof symbol)
;; 用 a-ftree 中所有的蓝眼睛祖先构造一个表
(define (all-blue-eyed-ancestors a-ftree0)
  (local (;; todo 是遇到过但还没有处理过的家谱树的表
          (define (all-a a-ftree todo)
            (cond
              [(empty? a-ftree)
               (cond
                 [(empty? todo) empty]
                 [else (all-a (first todo) (rest todo))])]
              [else
               (local ((define in-parents
                         (all-a (child-father a-ftree)
                                (cons (child-mother a-ftree) todo))))
                 (cond
                   [(symbol=? (child-eyes a-ftree) 'blue)
                    (cons (child-name a-ftree) in-parents)]
                   [else in-parents]))])))
    (all-a a-ftree0 empty)))
```

图 32.3 收集蓝眼睛的祖先（第二个带累积器的版本）

图 32.3 给出了第二个带累积器的 *all-blue-eyed-ancestors* 的完整定义。其中的辅助函数就是最常见的递归函数定义，它的第一和第二个 cond 子句分别包括了一处对 *all-a* 的调用。也就是说，这个函数定义并不与它主要输入（即家谱树）的数据定义相匹配。虽然这样的函数可以从遵照设计诀窍得出的可运行的函数开始，按照严谨的一系列开发步骤得出，但是我们不可能一开始就得出这个函数。

在处理程序表示法的程序中，累积器的使用是非常常见的。我们在 14.4 节中遇到过这种形式的数据，与家谱树类似，它们也有着复杂的数据定义。在第 18 章，我们也讨论过关于变量以及其相互联系的概念。下面习题中的一些简单函数，涉及了参数辖域、变量绑定等概念。

习题

习题 32.1.1 涉及如下所示的 Scheme 表达式子集（其中只包括极少数的 Scheme 表达式）的数据表示法：

```
<exp>   =   <var> | (lambda (<var>) <exp>) | (<exp> <exp>)
```

这个子集只包含了三种表达式：变量、带有一个参数的函数以及函数调用。例如：

```
1. (lambda (x) y)

2. ((lambda (x) x)
    (lambda (x) x))
3. (((lambda (y)
      (lambda (x)
       y))
     (lambda (z) z))
    (lambda (w) w))
```

用符号代表变量，把这些类型的数据称为 *Lam*。

回忆一下，lambda 表达式是没有名字的函数，因而它们在主体中绑定自己的参数。换句话说，lambda 表达式的参数的辖域就是它的主体。说明上述例子中每个绑定变量的辖域。用箭头连接所有的绑定变

和被绑定的变量。

如果某个变量出现在一个表达式之中，但是并没有对应的绑定变量，这个变量就被称为是自由的。构造一个既包含自由变量又包含绑定变量的表达式。

习题32.1.2 设计函数

```
;; free-or-bound : Lam -> Lam
;; 根据变量是否被绑定，把 a-lam 中
;; 所有未绑定的变量都替换成'free 或者'bound。
(define (free-or-bound a-lam) ...)
```

其中的 *Lam* 就是习题32.1.1中的表达式表示法。

习题32.1.3 设计函数

```
;; unique-binding : Lam -> Lam
;; 替换绑定变量以及它们所对应的绑定物的名字，
;; 使得一个名字在绑定中出现两次
(define (unique-binding a-lam) ...)
```

其中的 *Lam* 就是习题32.1.1中的表达式表示法。

提示：函数 gensym 从给定的符号产生一个新的、唯一的符号，该符号与其他程序中的符号都不相同，而且与 gensym 以前产生的符号也不相同。

使用这一节中提到的技术改进这个函数。提示：累积器把老的参数名和新的参数名联系在一起。

32.2 补充练习：传教士和食人者问题

有时函数要同时处理同一个类型的许多条数据，而累积器是某个复合数据的一部分。下面的故事正好描述了这样一个问题：

从前，有三个吃人肉的野蛮人为三个传教士带路穿越丛林前往最近的传教区。经过一段时间，他们来到了一条很宽的河流前，河中充满了各种会致人死地的蛇和鱼。过河的唯一方法就是乘船。幸运的是，很快他们就在河边找到了一条划船，上面还有两把浆。不幸的是，这条船太小了，一次只能乘坐两个人。更糟糕的是，这条河很宽，要想让船从对岸回来，唯一的方法就是由人把它划回来。

因为传教士并不信任食人者，所以他们必须制定一个计划，使得他们六个人能够安全地渡过河流。传教士担心的是，在任何地方，只要食人者的数量比传教士多，他们就可能会杀掉传教士，并且吃掉。所以，传教士制定的计划必须保证，在任何时刻，在河的任何一侧的岸上，都不会出现传教士的人数少于食人者的情况。不过，在其他方面，食人者是可以信任的，他们只是不会放弃任何可能的食物，就好像传教士不会放弃任何可能的皈依者一样。

万幸的是，其中一位传教士学习过 Scheme，他知道如何来解决这个问题。

虽然我们可以手工解决这个问题，但是用 Scheme 函数来求解这个问题更为有趣，也更具一般性。如果以后出现类似的问题，只是其中传教士的人数不同、食人者的人数不同或者是船的大小不同，还可以使用同样的函数来解决这个问题。

与处理其他问题一样，我们首先考虑如何用数据语言表达问题，并研究如何用程序设计语言描述现实世界中的某些行为。下面是两个关于数据表达的基本常数：

```
(define MC 3)
(define BOAT-CAPACITY 2)
```

用这些常数来给出函数。

习题

习题 32.2.1　给出问题状态的数据表示，状态应当记录河两岸传教士和食人者的数目，以及船的位置。下面是状态的一个图形表达：

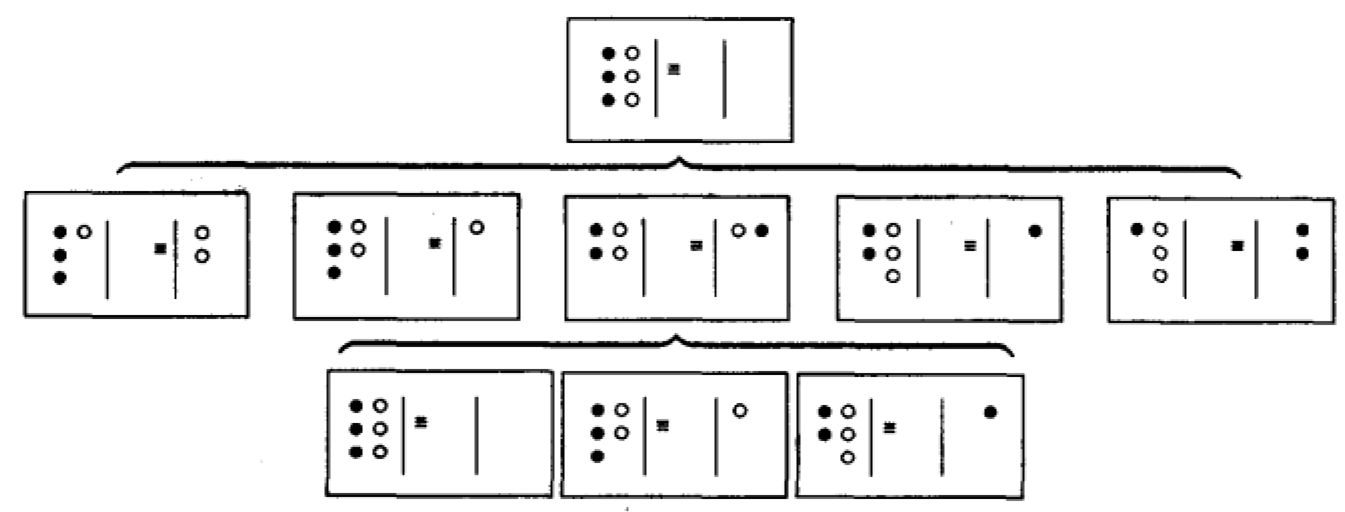
两条直线代表了河流，黑色和白色的点分别代表了传教士和食人者，黑色的矩形代表了小船。确定游戏的初始状态和最终状态。

习题 32.2.2　设计船的装载的数据表示法。定义 *BOAT-LOADS* 以表示所有可能的船的装载方式的表。

开发函数 *make-BOAT-LOADS*，该函数读入船的最大装载量，构造所有可能的船的装载方式的表。

一种有计划地处理搜索问题的方法是，生成目前所遇到的所有状态的可能后续，筛选出我们所感兴趣的，然后再用这些状态搜索。后续状态是指从目前状态通过合理的转化可以到达的状态，例如，游戏中所允许的一步移动，小船的一次航行，等等。

下面是这个问题的状态图形演示：



第一行中的初始状态有五个可能的后续状态，每种状态对应一种可能的船的装载。第二行给出了这五种状态。其中的两种后续是不合法的，因为这样的话某一侧岸上的食人者的数量就会超过传教士。其中一个合法的后续状态是，一个传教士和一个食人者到达了河的右岸；这个状态又有三种后续状态。下面的习题涉及产生后续状态以及筛选感兴趣的状态的练习。

习题

测试：把所有的测试表示为布尔值表达式，如果计算出的结果就是期望值，产生 true，反之产生 false。

习题 32.2.3　设计一个函数，读入一个状态，返回所有可能的后续状态，即所有让小船从河的一侧走到对岸所到达的状态。设计一个一般化的函数，读入状态表，返回从这些状态出发一次渡河所能到达的状态的表。

习题 32.2.4　设计一个函数，判断给定的状态是否是合法的。如果某个状态包含了正确数量的传教士和食人者，并且在河的两岸传教士都是安全的，那么这个状态就是合法的。设计一个一般化的函数，读入状态表，返回合法状态构成的子表（即选出所有的合法状态）。

习题 32.2.5　设计一个函数，读入一个状态，判断它是不是最终状态。设计一个一般化的函数，读入状态表，返回最终状态构成的子表。

我们已经开发的函数可以产生一个状态表的后续状态，还可以探测当前所到达的任何一个状态是否是合法的。现在我们可以开发判断是否可以把传教士和食人者运送到河对岸的函数了。

习题

习题 32.2.6　设计函数 *mc-solvable?*，该函数读入一个状态表，产生所有后续状态的表，直到它找到一个最终状态为止。当找到最终状态时，函数返回 true。

习题 32.2.7　设计函数 *mc-solution*，该函数是 *mc-solvable?*的改进。如果给定的传教士和食人者问题有解，当找到一个解时，函数不只产生 true，而是返回过河动作的表。

提示：修改状态表示法，使其记住到达特定状态所执行的过河动作。对于初始状态来说，这个表为空；对于最终状态，它就是所求的解。

习题 32.2.8　一系列的渡河动作可能把传教士和食人者带回到初始状态（或者其他以前到达过的状态）。这一系列的动作可能包含两个、四个或是更多的动作。简而言之，这个“游戏”可能包含循环，试构造一个循环的例子。

函数 *mc-solution* 产生所有可以到达的状态，如在该函数产生需要八次动作到达的状态之前，它先生成七次渡河可达的状态。这样，我们就无需担心在求解中出现循环。为什么？

修改解决方案，使得通过循环到达的状态也是不合法的。

注意：这表明，在状态表达中的累积器有两种用途。

32.3　补充练习：单人跳棋

单人跳棋是一种一个人玩的棋类游戏，棋中棋盘可以有着不同的形状。下面是最简单的棋盘形状：

```
   ⊙
  ⊙ ⊙
 ⊙ ○ ⊙
⊙ ⊙ ⊙ ⊙
```

当中没有黑点的圆圈表示一个未被占据的洞；其他的圆圈是包含棋子的洞。

游戏的目的是一个接一个地把棋子去除，直到剩下一个棋子为止。去除某个棋子的条件是，与它相邻的某个洞是空的，并且在相反方向的洞中有一个棋子。在这种情况下，第二个棋子可以跳过第一个棋子，并且把第一个棋子去除。考虑如下的布局：

```
2 →  ⊙
    ⊙ ⊙
1 →⊙ ⊙ ○
  ⊙ ⊙ ⊙ ⊙
```

在这种布局下，标号为 1 和 2 的棋子是可以跳动的。如果游戏者移动标号为 **2** 的棋子，接下来棋盘就变为

```
   ○
  ⊙ ○
 ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙
```

某些布局是死局。作为一个简单的例子，考虑第一个棋盘的布局。它的空洞位于棋盘的中央。这样就没有一个棋子可以跳跃，因为没有任何两个位于一条直线（包括水平线、垂直线和斜线）上的棋子可以跳过对方而跃入空洞。遇到死局时游戏者只能停下来，或者可以通过撤销移动，由原路返回，再试探

其他的跳跃。

习题

习题 32.3.1 设计三角形单人棋子游戏棋盘的表示法。设计棋子移动的数据表示法，假设棋子可以沿水平线、垂直线和斜线移动。

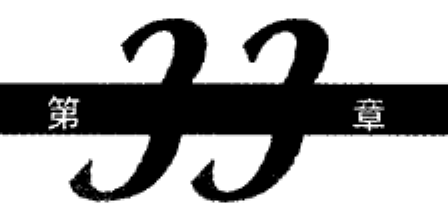
提示：（1）棋盘至少要有四行，因为如果只有三行或更少，游戏就无法进行。尽管如此，开发数据定义时不要考虑这个限制；（2）使用选用的数据表示翻译前面的例子。

习题 32.3.2 设计一个函数，该函数读入一个棋盘以及棋盘上棋子的分布，判断是否存在某个棋子可以跳跃。一般称可以跳跃的棋子是活动的。设计一个函数，读入一个布局以及棋盘上一个活动的棋子位置，创建下一个布局。

习题 32.3.3 开发函数 *solitaire*，求解不同大小的等边三角形棋盘上的单人跳棋游戏。这个函数应当读入一个布局，如果给定的问题不可解，它返回 false，否则，它应当生成一个移动的表，按照这个表给出的顺序移动棋子，就可以解决给定的单人跳棋游戏。

以布尔值表达式的形式，给出所有这些函数的测试。





非精确数的本质

计算机使用固定长度的数据块表示和处理信息。最早的计算机是被用来进行数值计算的，早期的计算机工程师使用了一种以固定长度的块表示数的方法。程序设计语言必须解决固定长度的数据表示和真正的数值之间的差距。另外由于使用数的基于硬件的表示法可以使程序运行的速度达到最高，所以大多数程序设计语言的设计者和程序都采用与硬件一致的方法来表示数。

本章详细地阐述了固定长度的数的表示法以及它的重要性。第一节介绍了一种具体的固定长度的数的表示法，讨论它的含义，说明如何在其上进行计算；第二节和第三节分别举例说明了固定长度的数关于算术运算的两个基本问题：上溢出和下溢出。

33.1 固定长度的数的算术运算

假设可以使用四个数位来表示一个数，如果要表示的是自然数，可以表示的范围就是从 0 到 9999。用同样的位数，也可以表示 0 到 1 之间的 10000 个分数。无论是哪一种情况，使用这四个位都只能表示相当小的一个范围内的数，而这对大部分的科学或是商业计算来说都是不实用的。

作为代替的方法，我们可以使用另外一种记数法来表示很大范围内的数。例如，在科学上，我们就采用科学记数法，使用两个部分来表示一个数：

1. 尾数；
2. 指数。

对于纯粹的科学记数法来说，基数是在 0 和 9 之间的数。放宽这个条件，把数记为：

$$m \cdot 10^e$$

其中 m 是尾数而 e 是指数。例如，用这种体制表示 1200 的一种表示是：

$$120 \cdot 10^1;$$

另一种表示是：

$$12 \cdot 10^2.$$

通常，一个数会有好几种等价的尾数-指数表示。

也可以使用负的指数，也就是使用带符号的指数来表示分数。例如，

$$1 \cdot 10^{-2}$$

代表了

$$\frac{1}{100}.$$

按照这种记数法，分数也有着多种不同的表示法。

要使用尾数-指数形式的记数法来解决记数的问题，还必须确定要使用多少数位来表示尾数和指数。这里分别使用两个数位来表示尾数和指数，另外还使用了指数符号，当然使用其他数位也是可行的。确定了位数，我们仍然可以把 0 表示为：

$$0 \cdot 10^{0}.$$

此时我们可以表示的最大数是：

$$99 \cdot 10^{99},$$

即，在 99 后面放上 99 个 0。如果除了使用正指数外还使用负指数，则可以表示：

$$01 \cdot 10^{-99},$$

这是一个接近于 0 的小数。这样，使用四个数位和一个符号，就可以表示一个比原先范围大得多的数。但是，这种改进的记数法也有它自身的问题。

```
;; create-inex : N N N -> inex
;; 在检查参数的正确性后创建 inex（非精确数）的实例
(define (create-inex m s e)
  (cond
    [(and (<= 0 m 99) (<= 0 e 99) (or (= s +1) (= s -1)))
     (make-inex m s e)]
    [else
     (error 'make-inex "(<= 0 m 99), +1 or -1, (<= 0 e 99) expected")]))

;; inex->number : inex -> number
;; 把 inex（非精确数）转化成与它相等的数值
(define (inex->number an-inex)
  (* (inex-mantissa an-inex)
     (expt 10 (* (inex-sign an-inex) (inex-exponent an-inex)))))
```

图 33.1 关非精确表示的函数

要理解这些问题，最好的方法是使用一种固定的表示方法对数进行实验。不妨用以下结构体来表示固定长度的数：

```
(define-struct inex (mantissa sign exponent))
```

该结构体包含三个字段，第一个字段和最后一个字段分别是数的尾数和指数，字段 sign 的值是+1 或者-1，表示指数的符号，使用这个符号字段我们可以表示 0 和 1 之间的数。

数据定义如下：

> *inex* 是结构体：
>
> (make-inex *m s e*)
>
> 其中 *m* 和 *e* 是位于[*0*，*99*]之中的自然数，*s* 是+1 或者-1。

因为 *inex* 的字段的条件是如此严格，所以我们不得不使用 *create-inex* 函数来建立这种结构体。图 33.1

给出了函数 *create-inex* 的定义，它是一个一般化的构造器，或者说是一个带检查的构造器（参见第 7.5 节）。图 33.1 还定义了函数 *inex->number*，它的功能是按照新的记数法把 inex 转换成数。

现在，让我们把前述的例子（1200），转换成 Scheme 表示法：

```
(create-inex 12 +1 2)
```

不过，另一种表示方法，120 10^1，在我们的 Scheme 世界中是不合法的。如果计算

```
(create-inex 120 +1 1)
```

的值，因为参数并不符合数据合约，我们会得到一个错误消息。另外，对于其他一些数，我们可以找到两个与之相等的 *inex*。一个这样的例子就是，0.0000000000000000005。我们可以把它表示为：

```
(create-inex 50 -1 20)
```

或者

```
(create-inex 5 -1 19)
```

请使用 *inex->number* 证实这两个表示法是等价的。

inex 数的范围是巨大的：

```
(define MAX-POSITIVE (make-inex 99 +1 99))
(define MIN-POSITIVE (make-inex 1 -1 99))
```

也就是说，我们既可以表示在原来的记数法中长达 101 位的数，也可以表示小于 1，最小可以达到 1 除以 10……0(总共有 99 个零)的正分数。可是，这种现象是骗人的。并不是所有的在 0 和 *MAX-POSITIVE* 之间的实数都可以被转化成 *inex* 结构体。精确地说，任何小于

$$10^{-99}$$

的正数都没有相应的 *inex* 结构体。同样，*inex* 表示法在所表示的数中间含有缺口。例如，

```
(create-inex 12 +1 2)
```

的后续是：

```
(create-inex 13 +1 2)
```

前一个 *inex* 结构体对应 1200，后一个对应 1300。在这两个数之间的数，例如 1240 或 1260，要么被表示成 1200，要么被表示成 1300。标准的方法是通过四舍五入把一个数舍入到与它最接近的、可以被表示的数。简而言之，在使用某种表示法时，先必须对数学上的数进行近似。

最后还必须考虑对 *inex* 结构体的算术运算。两个指数相同的 *inex* 数相加意味着把尾数相加：

```
  (inex+ (create-inex 1 +1 0)
         (create-inex 2 +1 0))
= (create-inex 3 +1 0)
```

翻译成数学符号，就是：

$$\begin{array}{r} 1\cdot 10^0 \\ +2\cdot 10^0 \\ \hline 3\cdot 10^0 \end{array}$$

如果两个尾数的和产生了太多的数位，就必须另找另一种合适的表示法。考虑把

$$55\cdot 10^0$$

加到它自身的例子。数学上，我们得到

$$110\cdot 10^0$$

但是不能直接把这个数简单地转化为我们的表示法，因为 110>99。正确的补救方法是把结果表示为

$$11 \cdot 10^1$$

或者，翻译成 Scheme，我们必须保证 *inex+*是这样进行计算的：

```
(inex+ (create-inex 55 +1 0)
        (create-inex 55 +1 0))
= (create-inex 11 +1 1)
```

更一般地说，如果计算结果的尾数过大了，我们必须把它除以 10，同时把指数增加 1。

有时，计算的结果包含的尾数位数比我们能够表示的要多。这时，*inex+*必须把尾数近似为 *inex* 所能表达的最接近的数。例如：

```
(inex+ (create-inex 56 +1 0)
       (create-inex 56 +1 0)) /
= (create-inex 11 +1 1)
```

这对应于精确计算：

$$56 \cdot 10^0 + 56 \cdot 10^0 = (56+56) \cdot 10^0 = 112 \cdot 10^0$$

因为计算的结果包含了太多的尾数位，把其中的尾数除以 10，再取它的整数近似值，就可以得到近似的计算结果：

$$11 \cdot 10^1$$

事实上有多种近似使得非精确算术变得不精确，这就是一个例子。

也可以相乘两个用 inex 结构体表示的数。回忆一下，

$$\begin{aligned}&(a \cdot 10^n) \cdot (b \cdot 10^m)\\ =&(a \cdot b) \cdot 10^n \cdot 10^m\\ =&(a \cdot b) \cdot 10^{(n+m)}\end{aligned}$$

因而我们有：

$$2 \cdot 10^{+1} * 8 \cdot 10^{+10} = 16 \cdot 10^{+11}$$

或者，使用 Scheme 符号：

```
(inex* (create-inex 2 +1 4)
       (create-inex 8 +1 10))
= (make-inex 16 +1 14)
```

与加法类似，事情并非总是那么简单。如果结果中尾数的数位太多了，*inex**必须增加指数的值：

```
(inex* (create-inex 20 -1 1)
       (create-inex  5 +1 4))
= (create-inex 10 +1 4)
```

在处理的过程中，如果产生的尾数不能直接用 *inex* 结构体表示，*inex**会引入一个近似值：

```
(inex* (create-inex 27 -1 1)
```

```
    (create-inex  7 +1 4))
  = (create-inex 19 +1 4)
```

习题

习题 33.1.1　设计函数 *inex+*求取两个由 *inex* 表示的有着相同指数的数之和。该函数必须能够处理指数增加的情况。另外，如果计算的结果超过了 *inex* 所能表达的范围，应当产生错误消息。

挑战：扩展 *inex+*的功能，使得它可以处理指数相差 1 的输入：

```
(equal? (inex+ (create-inex 1 +1 0) (create-inex 1 -1 1))
        (create-inex 11 -1 1))
```

在阅读完下一节之前，不要试图处理更大范围的输入。

习题 33.1.2　设计函数 *inex**，相乘 *inex* 表示的数。该函数必须能够处理指数增加的情况。另外，如果计算的结果超过了 *inex* 的表达范围，它必须产生自己的错误消息。

习题 33.1.3　本节通过举例，说明了非精确数表示体系在表示实数时是怎样形成缺口的。例如，1240 被舍去尾数的最低位，表示为(*create-inex* 12 +1 2)。问题是，舍入误差可以积累。

设计函数 *add*，其功能是把 *n* 个#i1/185 加起来。(*add* 185)的计算结果是什么？它应该是什么？如果我们用一个很大的数去乘(*add* 185)的返回值，将会发生什么？

设计函数 *sub*，使用递归计算从其参数中减去多少个 1/185 后参数会变成 0。计算(*sub* 1)和(*sub* #i1)时，该函数分别递归调用了多少次？在第二种情况下发生了什么？为什么？

33.2 上 溢 出

虽然科学记数法的使用扩展了基于固定长度的数据块所能表示的数的范围，但是它还未能覆盖任意大的数。毕竟有些数实在是太大了，以至于不能用固定的长度表示。例如，

$$99 \cdot 10^{500}$$

就无法被表示，因为使用两个数位无法表示指数 500，况且尾数还可以无限制放大。

在计算个过程中，有可能产生超过我们的记数体系所能表示的数。例如，两个可以表示的数相加，却可能产生不能表示的数：

```
(inex+ (create-inex 50 +1 99)
       (create-inex 50 +1 99))
= (create-inex 100 +1 99)
```

显然违反了数据合约，而

```
(inex+ (create-inex 50 +1 99)
       (create-inex 50 +1 99))
= (create-inex 10 +1 100)
```

也违反了 *inex* 结构体的合约。当 *inex* 算术运算的结果超过了其表示能力，我们就说发生了上溢(出)。

如果发生了上溢出，一些语言实现会产生一个错误信号，并且停止计算，另一些语言实现则指定一个叫做无穷大的符号，用来表示所有超大的数，此后算术运算能够意识到无穷大的存在，并且传播它。

负数：如果我们的 *inex* 结构体的尾数也有一个符号字段，那么两个负数相加就有可能产生一个很大

的负数，以至于它也不能被表达。这种情况也被称作上溢，不过，为了强调两种上溢的不同，人们有时把它称作负方向的上溢。

习题

习题 33.2.1 DrScheme 的非精确数体系使用一个无穷大值来处理上溢。给出这样的 *n*，使得(expt #i10. *n*)仍然是非精确 Scheme 数，而(expt #i10. (+ *n* 1))是近似的无穷大。提示：使用一个函数来计算 *n*。

33.3 下溢出

作为问题的另一个方面，我们已经看到过不能用 *inex* 结构体来表示的小数。例如，10^{-500} 并不是 **0**，但是它比我们所能表示的最小的非零数还要小。当两个很小的数相乘时，计算的结果就会变得如此之小，以至于不能被放入 *inex* 结构体之中，这就发生了下溢（出）：

```
  (inex* (create-inex 1 -1 10)
         (create-inex 1 -1 99))
= (create-inex 1 -1 109)
```

此时发生了错误。

在下溢发生时，一些语言实现产生一个错误信号；另一些语言实现则用 **0** 来近似结果。用 **0** 来近似下溢与先前其他类型的近似有着本质的区别。在用(*create-inex* **12 +1 2**)近似 **1250** 时，忽略尾数的有效位，但是仍然保留了一个非零的尾数。近似的结果与原来要表示的数的相差不超过 **10%**。然而，对下溢进行近似意味着忽略整个尾数，近似的结果并不在真实值的某个可预知的百分比范围之内。

习题

习题 33.3.1 DrScheme 的非精确数体系使用#i0 来近似下溢。给出最小的整数 *n*，使得(expt #i10. *n*)仍然是非精确 Scheme 数，而(expt #i10. (-*n* 1))是近似的 0。提示：使用一个函数来计算 *n*。

33.4 DrScheme 数

大多数的程序设计语言只支持整数以及实数的非精确表示（以及相应的算术运算）。与此不同，Scheme 既支持非精确数及其算术运算，又支持精确数及其算术运算，当然，内部表示是 2 进制，而不是 10 进制。

正如第 2 章所提到的，DrScheme 把所有的数理解为精确的有理数，除非它们以#i 开头。不过，一些数值操作会产生非精确数。纯粹的 Scheme（在 DrScheme 中被称为 Full Scheme）把所有带小数点的数理解为非精确的数[1]；另外在显示非精确数时也使用小数点符号，表示这种数是不精确的，有可能与真实的结果不同。

这样，Scheme 程序员就可以根据需要来选择使用精确数或者非精确数。例如，金融财政方面的数总应是精确的；对这类数据进行操作的算术运算应当尽可能精确。不过，对于有些问题，我们可能并不想

[1] 我们可以强迫 Full Scheme 通过将数的前面加前缀#e 而把带点的数解析为精确的数。

花费额外的时间来求得精确的解。例如，科学计算就是主要的一类使用非精确数的例子。在这种情况下，我们可能转而使用非精确数。

数值分析：　在使用非精确数及其算术的时候，自然地，我们要问，程序给出的解和真正的解有多少差异。在过去的几十年中，对这个复杂问题的研究发展成了数值分析，在应用数学和计算机科学领域，数值分析已经成为一门独立的学科。

习题

习题 33.4.1　分别使用 Full Scheme（无论是它的哪个变体）和 Intermediate Student Scheme 计算

```
(expt 1.001 1e-12)
```

解释观察到的结果。

习题 33.4.2　开发函数 *my-expt*，求某个整数的幂（指数函数）。用这个函数进行如下的实验，把

```
(define inex (+ 1 #i1e-12))
(define exac (+ 1 1e-12))
```

加入到 Definitions 窗口中。考虑(*my-expt* 30 *inex*)是什么？(*my-expt* 30 *exac*)又是什么？哪个答案更有用？

习题 33.4.3　如果求两个大小相差很多的非精确数的和，得到的结果可能就是其中大的那个数。例如，如果只使用 15 个有效位，那么在相加两个大小差距大于 10^{16} 倍的数时，就会遇问题：

$$10 \cdot 10^{16} + 1 = 1.0000000000000001 \cdot 10^{16},$$

如果系统只支持 15 位数的表示，最接近的答案就是 10^{16}。似乎这并不是太糟糕，毕竟，10^{16}（一亿亿）相差一，与正确的结果相比已经足够精确了。不幸的是，这类误差可能积累起来，造成大的问题。

考虑如下的非精确数的表：

```
(define JANUS
  (list #i31
        #i2e+34
        #i-1.2345678901235e+80
        #i2749
        #i-2939234
        #i-2e+33
        #i3.2e+270
        #i17
        #i-2.4e+270
        #i4.2344294738446e+170
        #i1
        #i-8e+269
        #i0
        #i99))
```

分别求出(*sum JANUS*)和(*sum* (reverse *JANUS*))的值，解释它们的差别并思考问题：我们能信任计算机吗？



第七部分

改变变量的状态



函数的记忆



无论调用一个函数多少次，只要使用相同的参数，总会得到相同的结果。即使是带累积器的函数，只要累积器相同，返回值也将相同。函数只是不能记住过去的调用结果。

可是，许多函数有时必须记住以往调用中的某些数据。回忆一下，典型的程序是由若干个函数组成的。过去，我们总是假设程序中有一个主函数，其他所有的函数都是辅助函数，辅助函数对用户来说都是不可见的。然而，在某些情况下，用户可能会要求一个程序提供不止一种服务，而每种服务都使用一个函数实现。当一个程序向用户提供不止一个服务的时候；或者为了方便，函数使用了一个图形用户界面，此时函数必须要有记忆能力。

要在理论上掌握这一点比较困难，所以我们先来研究一些例子。第一个例子是通讯录电话号码管理程序。标准的通讯录软件至少提供两种服务：

1. 查找某个人的电话号码；
2. 向通讯录中添加一个人和他的电话号码。

按照原则，程序向用户提供两个函数。用户可以在 DrScheme 的 Interactions 窗口中用适当的数据调用这些函数；也可以使用一个包括文本框和按钮的图形用户界面，这时用户就不需要了解编程知识了。图 34.1 给出了这样的一个界面。

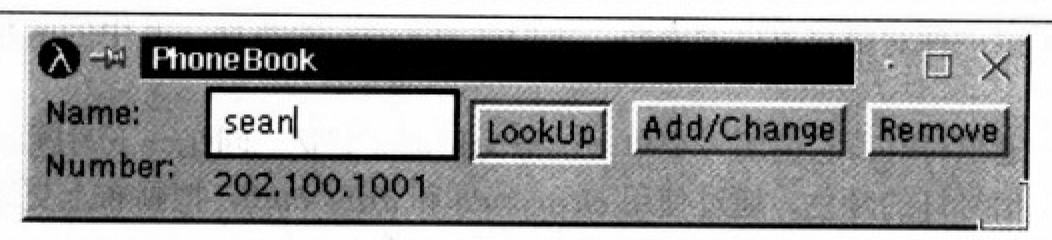


图 34.1　一个电话本 GUT

这两个服务大致对应于两个函数：

```
;; lookup : list-of-symbol-number-pairs symbol -> number or false
;; 在通信录中查找与 name 相关联的电话号码
;; 如果找不到 name，函数返回 false
(define (lookup pb name) ...)

;; add-to-address-book : symbol number -> void
;; 把名字和电话号码添加到通讯录中
(define (add-to-address-book name number) ...)

(define ADDRESS-BOOK
  (list (list 'Adam 1)
        (list 'Eve 2)))
```

上面我们引入了一个变量定义，用来保存名字-电话号码表。

第一个函数是我们最早遇到的一个递归函数的变体。用户把该函数作用于一个名字-电话号码表（例如 *ADDRESS-BOOK*）和一个名字。如果表中存在这个名字，函数就返回相应的号码，否则，就返回 false。第二个函数与我们所看到过的函数有着本质的不同。用户把该函数作用于一个名字和一个号码：以后，任何使用这个名字进行的查找都会返回对应的号码。

考虑如下 DrScheme 交互：

```
> (lookup ADDRESS-BOOK 'Adam)
1
> (lookup ADDRESS-BOOK 'Dawn)
false
> (add-to-address-book 'Dawn 4)
> (lookup ADDRESS-BOOK 'Dawn)
4
```

第一次交互确认'Adam 的电话号码是 1，第二次交互表明我们并不知道'Dawn 的电话号码，第三次交互把'Dawn 的电话号码 4 添加到 *ADDRESS-BOOK*，最后一次交互表明，与以前完全一样的 *lookup* 调用现在返回了所需的电话号码。

过去，要产生这样的效果，惟一的方法就是编辑 *ADDRESS-BOOK* 的定义。但是，我们并不希望用户来编辑程序。事实上，它们应该没有权力访问我们的程序。所以，我们应该提供一个函数界面，允许用户进行这种修改，甚至更进一步，实现如图 34.1 的图形界面，这时与上述交互等价的对话应该是这样的：

1. 在文本框中输入 Adam，单击 Lookup 按钮，于是“1”就出现在下方的文本框中。
2. 在文本框中输入 Dawn，单击 Lookup 按钮，于是下方的文本框中出现一条消息，表明没有要找的号码。
3. 把该条消息改成“4”，再单击 Add 按钮。
4. 去除下方文本框中的“4”，再单击 Lookup，“4”就又出现了。

简而言之，要提供一种对用户方便的界面，我们必须开发一个程序，其中的函数知道其他函数的调用历史。

第二个例子是交通信号灯，它说明一个单一函数为何需要记忆。回忆一下习题 6.2.5 中的 *next* 函数，该函数读入当前交通信号灯的颜色，通过使用 *clear-bulb* 和 *draw-bulb*，把画布上交通信号灯的状态改变为下一个状态，它的返回值是信号灯的下一种颜色。

如果某个用户要改变四次信号灯的状态，他就必须在 Interactions 窗口中输入：

(*next* (*next* (*next* (*next* 'red))))

不过，一个更方便的方法是使用包含按钮的用户界面，用户可通过单击按钮改变信号灯的状态。

提供一个按钮意味着提供一个回调函数(call-back function)，该回调函数要以某种方式得知当前信号灯的状态，并改变之。我们把这个函数也称作 *next*，并假定它没有参数。使用这个函数，一组想象中的交互是：

```
> (next)
> (next)
> (next)
```

每一次调用 *next*，函数都返回不可见的值，并在画布上模拟交通灯的状态改变。换一种说法，交通灯在图 34.2 所示的几种状态中循环。等价地说，用户按三次“NEXT”按钮，将三次调用 *next* 函数，从而产生相同的视觉效果。要实现这种效果，*next* 的当前调用必须能影响将来的调用

最后一个例子即 6.7 节中的刽子手游戏。这个游戏程序要求我们开发三个函数：*make-word*、*reveal* 和 *draw-next-part*。通过计算

```
(hangman make-word reveal draw-next-part)
```

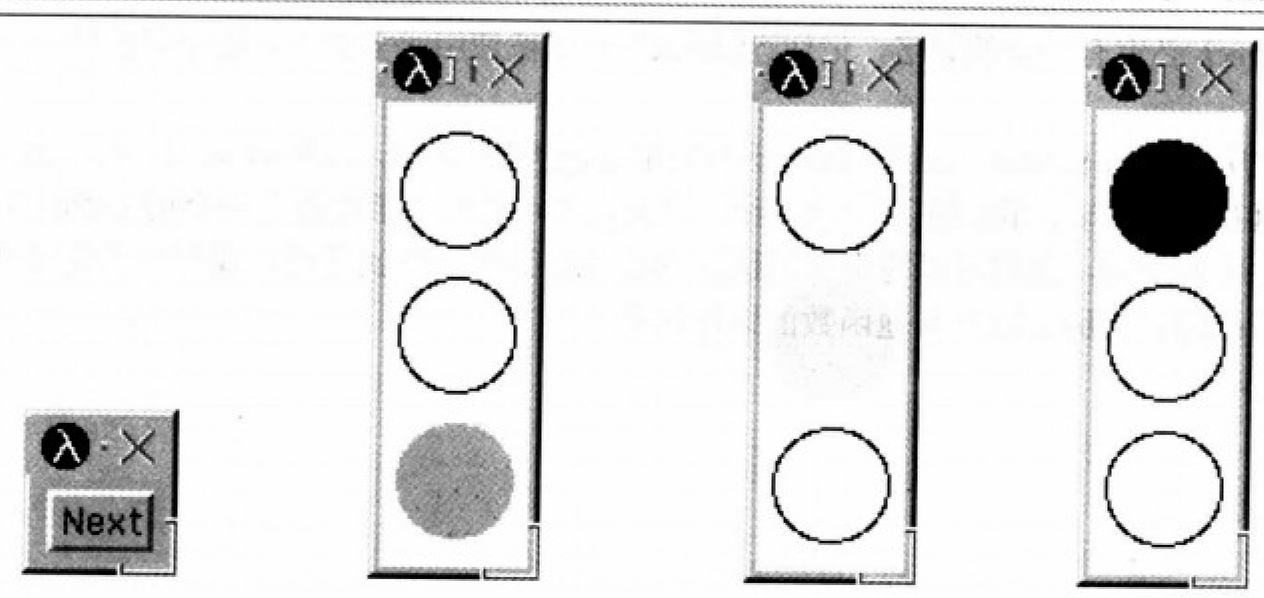


图 34.2　一个交通信号灯的三个阶段以及 GUT

开始游戏。游戏先挑选一个单词，建立如图 34.3 下方所示的图形用户界面，再绘制出图 34.3 左上方的图形。接着，玩家可以从 GUI 的菜单中选择一个字母，并单击“Check”按钮检查该字母是否在单词中出现。如果单词中存在该字母，*hangman* 函数就显示出该字母出现的位置；如果该字母不存在，就使用 *draw-next-part* 函数绘制图像。玩家所作的猜测越不准确，图像中出现的线条就越多（参见图 34.3 上部）。

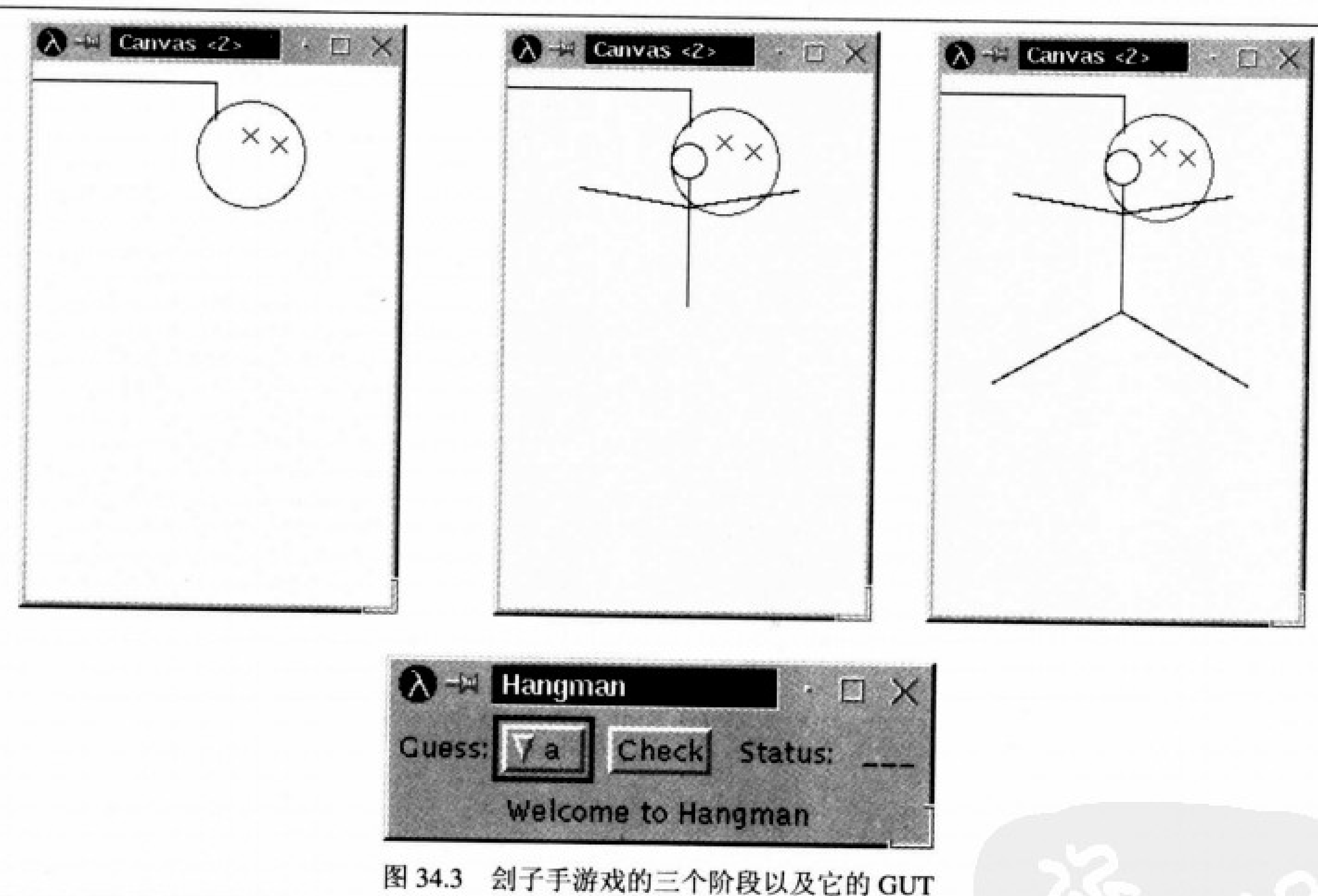


图 34.3　刽子手游戏的三个阶段以及它的 GUT

描述说明，教学软件包中的 *hangman* 函数使用一个回调函数来实现“Check”按钮的功能。我们把这个函数称为 *check*，该函数读入一个字母，如果检查发现这是单词中原先未知的字母，函数就返回 true：

```
> (check 'b)
true
```

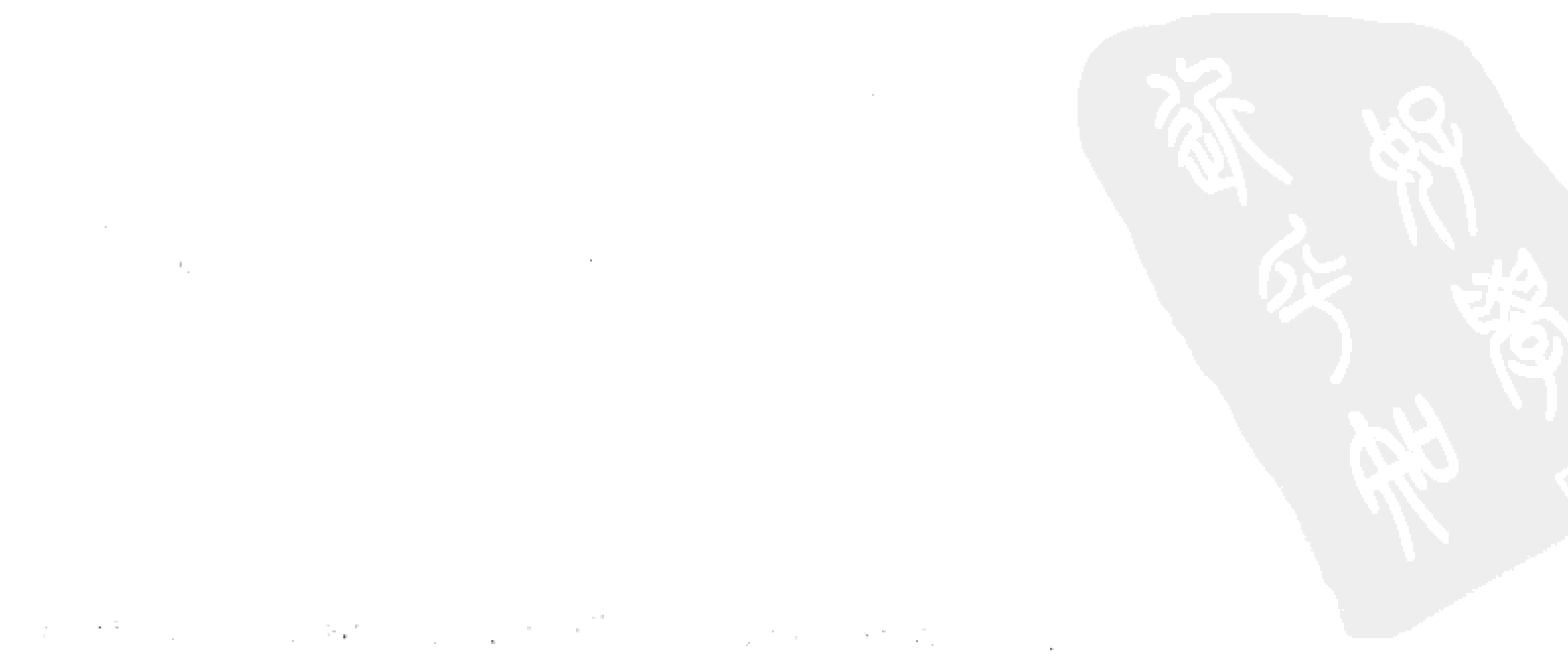
否则，返回 **false** 表明玩家这次猜测失败：

```
> (check 'b)
false
```

在这种情况下，*check* 还将调用 *draw-next-part*，绘制图像中的另一个部分。当然，要做到这一点，

hangman 和 *check* 必定有一定的记忆，记住“Check”按钮被使用了多少次，还要记住读入的猜测有多少次是错误的。

使用目前所掌握的 Scheme 知识，我们还不能定义出类似 *add-to-address-book*、*next* 或 *check* 这样的函数。为了填补这个空缺，我们在下一章介绍 set!表达式，它允许函数改变一个用 define 定义的变量的值。使用这种新的结构，我们可以写出有记忆的 Scheme 函数。也就是说，我们可以定义出一种函数，它知道一些自己的调用历史以及其他函数的调用历史。



对变量赋值



set!表达式也被称作赋值表达式，形状是：

```
(set! var exp)
```

它由一个变量和一个表达式组成，其中 *var* 称为赋值表达式的左部，exp 称为赋值表达式的右部。set!表达式的左部是一个使用 define 定义过的变量，当然它可以是在最外层定义，也可以是使用 local 表达式定义。set!表达式可以出现在任何一个表达式合法出现的地方。

所有 set!表达式的值都是相同的，也是不可见的，因此和计算不相关。与计算相关的是 set!表达式的效果。具体来说，在求一个 set!表达式的值时，第一步是求出 exp 的值，假定为 *V*；第二步是改变 *var* 的定义为

```
(define var V)
```

其效果是，从此以后，在计算的过程中，任何对 *var* 的访问都会把 *var* 替换成 *V*，而以前的值则被丢弃。

要真正理解赋值的本质是很困难的，所以我们先来考虑一个简单的例子。

35.1 简单的、能工作的赋值

考虑如下的定义和表达式：

```
(define x 3)
(local ((define z (set! x (+ x 2))))
  x)
```

该定义表明 *x* 代表 3。而 local 表达式引入了 *z* 的定义，它的主体是 *x*，在过去，这个 local 表达式的值会是 3。现在，使用了 set!，这一点就不再成立了。要理解发生了什么，必须一步一步计算，直到得到最终的答案为止。

计算的第一步是提升 local 表达式：

```
(define x 3)
(define z (set! x (+ x 2)))
x
```

接下来必须确定(set! *x* (+ *x* 2))的值。按照 set!的含义，我们必须计算赋值表达式的右部：

```
(define x 3)
(define z (set! x 5))
x
```

因为 *x* 的当前值是 3，所以这个值是 5。

计算表明，set!表达式的效果是改变赋值表达式左部变量的值。在例子中，这意味着从此以后，x 不再是 3 而是 5 了。要表示这个改变，最好的方法是将 x 的定义修改为两步：

```
(define x 5)
(define z (void))
x
```

set!的值是(void)，而(void)是一个不可见的值(invisible value)。将 set!表达式的值替换成不可见的值，表明计算已经完成。

到这里，我们可以明白这段表达式的计算结果是 5。第一个定义说明 x 现在表示 5，而最后一个表达式是 x。因此该函数计算出的值是 5。

习题

习题 35.1.1 考虑下列语句：

```
1.
(set! x 5)
2.
(define x 3)
(set! (+ x 1) 5)
3.
(define x 3)
(define y 7)
(define z false)

(set! (z x y) 5)
```

哪些是语法上合法的程序？哪些是不合法的程序？

习题 35.1.2 计算下面的程序：

```
(define x 1)
(define y 1)
(local ((define u (set! x (+ x 1)))
    (define v (set! y (- y 1))))
  (* x y))
```

如果 set!不是语言的一个部分，local 表达式的返回值会是什么？也就是说，考虑如下定义右部被去除的程序框架：

```
(define x 1)
(define y 1)
(local ((define u ...)
    (define v ...))
  (* x y))
```

在引入 **set!**表达式之前，这个表达式会返回什么？

35.2　顺序计算表达式

上一节手工计算表明，对 *z* 的 local 定义的作用是，求 set!表达式的值，再把这个值“扔掉”。毕竟，set!真正的用途是改变一个定义的值，而不是返回一个值。这种情况在 Scheme 中非常普遍，所以 Sheme 提供了 begin 表达式：

```
(begin exp-1
       ...
       exp-n
      exp)
```

begin 表达式由关键字 begin 开头，后面跟着 $n+1$ 个表达式。计算 begin 表达式时，首先按顺序求出所有的表达式的值，然后把前 *n* 个值丢弃。最后一个表达式的值是整个 begin 表达式的值。一般来说，在 begin 表达式中，前 *n* 个子表达式的作用是改变某些定义，只有最后一个子表达式给出我们所关注的值。

现在可以用一个更短的表达式来重写第一个 set!例子：

```
(define x 3)
(begin (set! x (+ x 2))
       x)
```

begin 的使用不仅简化了程序，还引入了一种计算顺序。

手工计算还说明，set!表达式的计算过程带来了额外的时间约束或时间区间。更具体地说，计算是由两个部分组成的：赋值前的部分和赋值后的部分，而赋值会影响定义的状态。在我们介绍赋值语句之前，只要愿意，随时可以把变量替换成它的值，或者把一个函数调用替换成它的定义。现在，我们必须等到真正需要某个变量的值的时候才能执行这样的替换。

虽然语句执行的顺序总是计算的一个部分，但是 set!带来的时间约束却是新的概念。赋值操作“消灭”了当前的值，除非程序设计者能详细安排变量的赋值顺序，使用 set!可能会带来灾难性的后果，习题会详细说明这个问题。

习题

习题 35.2.1　手工计算如下程序：

```
(define x 1)
(define y 1)
(begin (set! x (+ x 1))
       (set! y (- y 1))
       (* x y))
```

考虑共有多少个不同的可辨别的时间区间？将其与

```
(define a 5)

(* (+ a 1) (- a 1)))
```

的计算进行比较。考虑嵌套对计算来说是否是一种顺序？加法与减法的顺序是否会影响计算的结果？

习题 35.2.2　手工计算如下程序：

```
(define x 3)
(define y 5)
(begin (set! x y)
       (set! y x)
       (list x y))
```

考虑共有多少个不同的可辨别的时间区间？计算：

```
(define x 3)
(define y 5)
(local ((define z x))
  (begin (set! x y)
         (set! y z)
         (list x y)))
```

无论初始值是什么，当两个 **set!**表达式被计算之后，是不是 *x* 的定义包含了 *y* 的初始值，而 *y* 的定义包含了 *x* 的初始值？就定义的时间和“值的毁灭”进行讨论，同时考虑这两个例子告诉了我们什么？

习题 35.2.3 手工计算如下表达式：

```
(define x 3)
(define y 5)

(begin
  (set! x y)
  (set! y (+ y 2))
  (set! x 3)
  (list x y))
```

在这个手工计算中，我们总共可以辨别出多少个时间区间？

35.3 赋值和函数

赋值也可以出现在函数的主体之中：

```
(define x 3)

(define y 5)

 (define (swap-x-y x0 y0)
  (begin
    (set! x y0)
(set! y x0)))

 (swap-x-y x y)
```

这里，函数 *swap-x-y* 读入两个值，执行两个 set!。

我们来看看计算是如何进行的。因为(*swap-x-y x y*)是一个函数调用，所以先需要计算参数的值，而

参数是普通变量，所以可以用它们的当前值代替：

```
(define x 3)

(define y 5)

(define (swap-x-y x0 y0)
  (begin
    (set! x y0)
(set! y x0)))

(swap-x-y 3 5)
```

接着使用通常的替换规则继续进行计算：

```
(define x 3)

(define y 5)

(define (swap-x-y x0 y0)
  (begin
    (set! x y0)
(set! y x0)))

(begin
  (set! x 5)
  (set! y 3))
```

换句话说，函数调用现在被替换为用 y 的当前值对 x 赋值以及用 x 的当前值对 y 赋值。接下来的两步完成函数名称所表示的工作：

```
(define x 5)

(define y 3)

(define (swap-x-y x0 y0)
  (begin
(set! x y0)
(set! y x0)))

(void)
```

因为最后一个表达式是 set!表达式，因此函数的返回值是不可见的。

概括来说，使用 set!的函数既有返回值，又有效果，不过返回值有可能是不可见的。

习题

习题 35.3.1 考虑如下的函数定义：

```
(define (f x y)
  (begin
    (set! x y)
```

```
    y))
```

试考虑在语法上，它合不合法？

习题 35.3.2 手工计算如下程序：

```
(define x 3)

(define (increase-x)
  (begin
    (set! x (+ x 1))
x))

(increase-x)
(increase-x)
(increase-x)
```

其结果是什么？*increase-x* 的效果是什么？

习题 35.3.3 手工计算如下程序：

```
(define x 0)

(define (switch-x)
  (begin
    (set! x (- x 1))
x))

(switch-x)
(switch-x)
(switch-x)
```

其结果是什么？*switch-x* 的效果是什么？

习题 35.3.4 手工计算如下程序：

```
(define x 0)

(define y 1)
(define (change-to-3 z)
  (begin
    (set! y 3)
z))

(change-to-3 x)
```

change-to-3 的效果是什么？它的返回值是什么？

35.4 第一个有用的例子

让我们来看一看图 35.1 中的定义。函数 *add-to-address-book* 读入一个符号和一个数，前者表示一个

人的名字，后者表示他的电话号码。函数的主体包含了一个 set!表达式，该表达式对 *address-book*（一个最外层的变量）赋值。函数 *lookup* 读入一个通讯录和一个人名，返回值是那个人的电话号码，如果 *address-book* 不包含他的名字，就返回 false。

```
(define address-book empty)

;; add-to-address-book : symbol number -> void
(define (add-to-address-book name phone)
  (set! address-book (cons (list name phone) address-book)))

;; lookup : symbol (listof (list symbol number)) -> number or false
;; 在 ab 中查找 name 的电话号码
(define (lookup name ab)
  (cond
    [(empty? ab) false]
    [else (cond
            [(symbol=? (first (first ab)) name)
             (second (first ab))]
            [else (lookup name (rest ab))])]))
```

图 35.1　基本的通讯录程序

使用 *lookup* 可以研究在 *add-to-address-book* 中 set! 表达式的效果。假设使用给定的定义计算(*lookup* 'Adam *address-book*)：

```
  (lookup 'Adam address-book)
= (lookup 'Adam empty)
= (cond
    [(empty? empty) false]
    [else ...])
= false
```

因为 *address-book* 是 empty，所以得到 false，而且这个计算相当简单。

现在在 Interactions 窗口中计算如下表达式：

```
(begin (add-to-address-book 'Adam 1)
       (add-to-address-book 'Eve 2)
       (add-to-address-book 'Chris 6145384))
```

第一个子表达式是一个普通的函数调用，所以，计算的第一步基于通常的替换规则[1]：

(define *address-book* empty)

```
(begin (set! address-book (cons (list 'Adam 1) address-book))
       (add-to-address-book 'Eve 2)
       (add-to-address-book 'Chris 6145384))
```

下一个要被计算的表达式是嵌套在 begin 表达式中的 set!表达式，特别是 set!的右部。cons 的第一个参数是一个值，第二个参数是一个变量，其当前值为 empty。由此，我们来看下一步会发生什么：

```
(define address-book empty)
```

[1] 因为计算不会影响函数定义，因而我们这里在计算中没有包含函定义。这样节省了时间和空间，但是使用起来一定要小心

```
(begin (set! address-book (cons (list 'Adam 1) empty))
       (add-to-address-book 'Eve 2)
       (add-to-address-book 'Chris 6145384))
```

现在，我们为计算 set!表达式做好了准备。具体地说，我们改变 *address-book* 的定义，使得这个变量现在代表(cons (list 'Adam 1) empty)：

```
(define address-book
  (cons (list 'Adam 1)
empty))

(begin (void)
        (add-to-address-book 'Eve 2)
        (add-to-address-book 'Chris 6145384))
```

begin 表达式会把其中不可见的值丢弃。

计算剩下的 *add-to-address-book* 调用，产生：

```
(define address-book
  (list (list 'Chris 6145384)
    (list 'Eve 2)
    (list 'Adam 1)))
 (void)
```

简而言之，这三个调用把 *address-book* 转变为由三个姓名一电话号码对组成的表。

现在如果在 Interactions 窗口中求(*lookup* 'Adam *address-book*)的值，可以得到 **1**：

```
  (lookup 'Adam address-book)
= (lookup 'Adam (list (list 'Chris 6145384)
                (list 'Eve 2)
                (list 'Adam 1))
= ...
= 1
```

与本节一开始的计算相比，我们可以看到随着时间的逝去 set!改变了 *address-book* 的含义，而 *add-to-address-book* 和 *lookup* 两个函数确实实现了我们在第 34 章中讨论的服务。下面习题将说明，在图形用户界面中，这两个函数的合作非常有用。

习题

习题 35.4.1 管理通讯录的软件一般还允许用户从通讯录中去除一些条目。设计函数

```
;; remove : symbol -> void
(define (remove name) ...)
```

该函数改变 *address-book*，使得以后所有对 *name* 的查找都返回 false。

习题 35.4.2 教学软件包 phone-book.ss 实现了基于 22.3 节所讨论的模型视图模式的图形用户界面。图 34.1 显示了该图形用户界面所提供的东西：

1. 一个文本框，用来输入人名；
2. 一个文本框，用来显示查找结果，以及输入电话号码；
3. 一个按钮，用来查找某个人名所对应的电话号码；

4．一个按钮，用来添加一个人名和电话号码；

5．一个按钮，用来删除一个人名以及其电话号码。

使用教学软件包中的 connect 函数，建立本节所讨论的函数（包括习题 35.4.1 中的函数）的 GUI。该函数的合约、用途说明和头部如下：

;; *model-T* = (*button% control-event%* -> true)

;; *connect : model-T model-T model-T* -> true

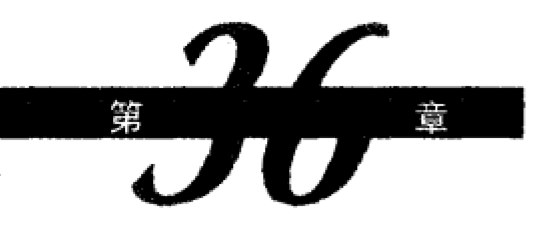
(define (*connect lookup-cb change-cb remove-cb*) ...)

也就是说，读入三个模型函数，并将其与 GUI 整合在一起，参数的名称指定了回调函数和按钮之间的对应。

模型函数可以使用(*name-control*)获得名称字段的内容，也可以用(*number-field*)获得号码字段的内容。



设计有记忆的函数



第 34 章提出了记忆函数的概念；第 35 章解释了变量定义与 **set!**如何一起达到记忆的效果。现在到了讨论设计有记忆的函数的时候了。

定义记忆函数需要三个重要的步骤：

1．确认确实需要记忆。

2．确定要记忆的数据。

3．理解哪项服务需要修改记忆，那项服务需要使用记忆。

第一步是必需的；一旦知道了某个程序需要记忆，就必须对记忆进行数据分析，也就是说，必须理解记忆的数据类型是什么；最后，必须仔细设计那些改变记忆的函数，只使用而不修改记忆的函数只需按照前面介绍过的设计诀窍进行设计就可以了。

36.1 对记忆的需求

如果希望几乎或完全不懂编程的用户也可以使用程序，程序就需要记忆。即使要求用户通过 DrScheme 的 Interactions 窗口使用程序，我们也必须对程序进行组织，使得每一种服务对应一个函数，而函数通过记忆相互合作。如果使用图形用户界面，那么必须把程序看作相互合作的函数的集合，而后者和窗口的各个小控件相联系。最后，即使是在物理设备（例如电梯、录像机等）中运行的函数，也必须以某种方式与设备相结合。简而言之，程序与世界的其余部分之间的界面决定了该程序是否需要记忆，以及它需要何种类型的记忆。

幸运的是，要辨认出何时程序需要记忆相对来说比较简单。我们已经讨论过，有两种情况。第一种情况是，程序向用户提供不止一种服务，每种服务对应一个函数。某个用户可能在 DrScheme 的 Interactions 窗口中调用这些函数，也可能用户操作一个图形用户界面，而函数通过用户操作被调用。第二种情况是，程序只提供单一的服务，而且这个服务用一个用户级的函数实现。但是，当该函数作用于同样的参数时，它可能会返回不同的答案。

对于这两种情况，我们分别以一个具体例子说明。管理通讯录的软件是第一种情况的一个经典例子。在第 34 章和第 35 章中，我们看到，一个函数向通讯录中添加元素，而另一个函数进行信息查找。显然，“添加服务”的使用影响了“查找服务”的使用，所以程序需要记忆。事实上，这种情况下的记忆对应于一种自然的物理对象：通讯录。在电子笔记本出现以前，人们都是使用通讯录来保存人员信息的。

接下来考虑仓库和管理员。管理员负责登记人们送入和取出的物品。一旦有东西被送入仓库，管理员就把它输入到一个帐簿中；对某个特定物品的查询就是搜索帐簿；一旦从仓库中取出东西，管理员就从帐簿中删除对应的数据。如果设计一个管理帐簿的程序，它就需要提供三种服务：输入物品，搜索帐簿，以及从帐簿中删除物品。当然，我们不能从仓库中取出一个不存在的东西，所以程序必须保证两种服务（指送入和取出物品）能正确地互相影响。在这个程序中，记忆就类似于仓库管理员所使用的帐簿，这也是一个物理的对象。

对于第二种情况，第 34 章中提到的交通信号灯就是一个经典的例子。回忆一下，程序 *next* 的描述表明，每一次对它的调用都会在画布上按照交通灯的规则重画图像。因为连续两次计算(*next*)会产生不同的效果，所以这个函数需要记忆。

另一个例子是 Scheme 函数 random，它读入大于 1 的自然数 *n*，返回一个在 0 和 *n*-1 之间的自然数。两次计算(random 10)，返回值可能相同，也可能不同。因此 random 的实现需要配备有记忆的函数。

一般来说，在分析问题说明时应该画出组织图。图 36.1 给出了两个例子，它们分别是通讯录管理程序和交通信号灯程序的组织图。组织图用一个矩形方框描述程序所提供的每一个服务。指向方框的箭头表示该服务所需的数据类型；从方框向外的箭头指定服务的输出。圆圈表示记忆。从圆圈到方框的箭头表示该服务使用记忆作为一个参数；从方框到圆圈的箭头表示该服务改变记忆。这两张图表明服务通常要使用记忆，并且要修改记忆。

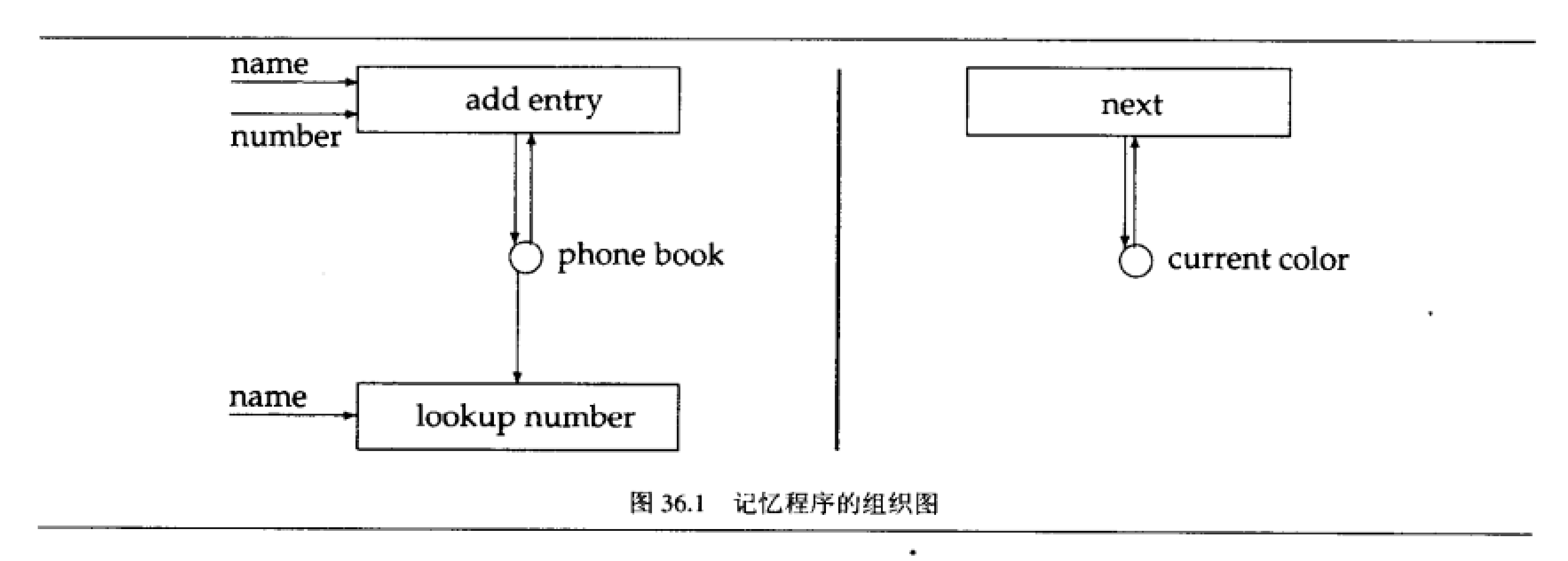


图 36.1　记忆程序的组织图

36.2　记忆与状态变量

记忆是用变量定义实现的。前面使用记忆的程序用一个单独的变量表示一个函数的记忆。原则上，只要一个变量就足够实现所有需要的记忆了，但是这通常是不方便的。一般来说，从记忆分析可以知道我们需要多少变量，以及哪项服务需要哪个变量。当记忆改变了，相应的变量就变成了一个新的值，或者换一种说法，变量声明的状态改变了，这反映了记忆随时间变化。因此，我们把实现记忆的变量称为状态变量。

在程序之中，每个服务对应于一个函数，而函数可能会使用辅助函数。改变程序的记忆的服务由对某个（或某些）状态变量使用 set!的函数实现。要理解函数是如何改变状态变量，需要先了解该变量表示的是哪种类型的值，以及它的用途是什么。换句话说，就像设计函数定义的合约与用途说明一样，我们必须设计状态变量的合约与用途说明。

我们来看通讯录和交通灯的例子。前一个例子有一个状态变量 *address-book*，它的目的是表示一个条目的表。要说明 *address-book* 只可以表示这样的表，我们加上如下的合约：

```
;; address-book : (listof (list symbol number))
;; 保存人名和电话号码的对
(define address-book empty)
```

既然它的定义是(listof *X*)，当然允许 empty 作为 *address-book* 的初始值。

从该状态变量的合约中可以推断出如下的赋值是没有意义的：

```
(set! address-book 5)
```

因为它把 *address-book* 的值置为 **5**，而 **5** 不是一个表，所以这个表达式违背了状态变量的合约。但是

```
(set! address-book empty)
```

是正确的，因为它把 address-book 设回初始值。第三个赋值是：

```
(set! address-book (cons (list 'Adam 1) address-book))
```

它帮助我们理解函数是怎样以一种有用的方式改变 *address-book* 的值。因为 *address-book* 代表了表的表，所以(cons (list 'Adam 1) *address-book*)构造了一个更长的、类型正确的表。因此，set!表达式正好改变了状态变量的值，使它代表(listof (list *symbol number*))类型中的另一个值。

控制交通信号灯的程序应当使用一个状态变量，用来保存信号灯的当前颜色。这个变量应当取如下三个值中的一个：'red、'green 或'yellow，这表明了它的数据定义：

TL-color 是 **'red**、**'green** 或 **'yellow** 三者之一。

下面是包括合约与用途说明的变量定义：

```
;; current-color : TL-color
;; 保存交通信号灯当前的颜色
(define current-color 'red)
```

同样，表达式

```
(set! current-color 5)
```

是没有意义的，因为 5 不是合约中所提到的三种合法符号中的一种。反之，

```
(set! current-color 'green)
```

是完全正确的。

赋值的右部并非一定要是一个值，或是一个几乎立刻就能产生一个值的表达式。在许多情况下，使用一个能够计算出值的函数也是合理的。下面就是一个函数，它能计算出交通信号灯的下一种颜色：

```
;; next-color : TL-color -> TL-color
;; 计算交通信号灯的下一种颜色
(define (next-color c)
  (cond
    [(symbol=? 'red c) 'green]
    [(symbol=? 'green c) 'yellow]
    [(symbol=? 'yellow c) 'red]))
```

使用这个函数可以写出一个赋值语句，正确地改变 *current-color* 的状态：

```
(set! current-color (next-color current-color))
```

因为 *current-color* 是三种合法符号中的一种，所以我们可以把 *next-color* 作用于 *current-color* 的值之上，该函数返回的值也是这三个符号中的一个，所以 *current-color* 的下一个状态也是正确的。

36.3 初始化记忆的函数

在完成了程序中状态变量的合约与用途说明的设计之后，我们应立即定义一个函数，把这些状态变量设置为合适的初始值。我们把这样的函数称为初始化函数。在程序运行时，初始化函数应当是第一个被执行的函数；程序也可以提供其他的方法来调用初始化函数。

对于我们的例子来说，初始化函数相当简单。下面是通讯录的初始化函数：

```
;; init-address-book : -> void
(define (init-address-book)
  (set! address-book empty))
```

交通灯的初始化函数也很普通：

```
;; init-traffic-light : -> void
(define (init-traffic-light)
  (set! current-color 'red))
```

遵从传统的工程规则，在启动设备时将状态设为最安全[1]，因此这里把 *current-color* 设为'red。

初看起来，初始化函数在函数中好像并没有添上许多东西，它们都把各自的状态变量设置为定义的值。然而，对这两个例子来说，很容易看出初始化函数还应当做另外一些有用的工作。例如，第一个例子中的初始化函数应当创建并显示图形用户界面；第二个例子中的初始化函数应当创建并显示画布以显示当前信号灯的状态。

36.4　改变记忆的函数

一旦有了状态变量和它们的初始化函数，我们就要把注意力集中到设计修改记忆的函数之上。与本书前面部分所介绍的函数不同，改变记忆的函数不仅读入并返回数据，它们还影响状态变量的定义，所以我们说这样的函数对状态变量会产生效果。

现在，我们来看看最基本的设计诀窍的各个阶段应该如何适应状态变量：

数据分析：即使是影响变量的状态的函数，也可以（或可能）读入并返回数据。因此仍然需要分析如何表示信息，如果必要的话，还要引入结构和数据的定义。

例如，交通信号灯的例子得益于 *TL-color* 的数据定义（参见前文）。

合约、用途和效果：第一个主要的改变是有关第二步的。除了要说明函数读入和返回的东西，还必须写明它影响了哪些变量，以及它是怎样影响这些变量的。函数对状态变量的效果必须与变量的用途说明一致。

再一次考虑交通灯的例子。我们需要一个函数，根据交通规则来转换信号灯的颜色，该函数检查变量 *current-color*，并影响它的状态。我们应该这样说明这个函数：

```
;; next : -> void
;; 效果：改变当前的颜色，从'green 变为'yellow，
;; 从'yellow 变为'red，从'red 变为'green
(define (next) ...)
```

该函数并不读入任何的数据，也不返回可见的值；在 Scheme 中，这个值被称作 *void*。在传统意义上，这个函数是没有意义的，所以它只有一个效果的说明（而没有其他的说明）。

下面是 *add-to-address-book* 的说明：

```
;; add-to-address-book : symbol number -> void
;; 效果：把(list name phone)添加到 address-book 的前部
(define (add-to-address-book name phone) ...)
```

从效果说明中可以看出，*address-book* 的定义是按照它的用途说明与合约被修改的。

程序例子：例子和以前一样重要，但明确地叙述例子变得更困难了。以前，我们必须开发例子，举例说明输入和输出的关系，但是，因为现在函数有了效果，所以我们还需要用例子来说明效果。

回过头来看我们所用的第一个例子，交通信号灯中的 *next* 函数，它影响一个状态变量：*current-color*。该变量只能代表三种符号中的一个，所以实际上我们可以用例子来表示其所有可能的效果：

```
;; 如果 current-color 是'green 时我们计算(next),
;; 那么其后 current-color 就是'yellow
```

[1] 当检测内部故障时也应该遵从这个建议，把状态设为最无害的状态，可是不幸的是，许多软件工程师并不遵从这一规则。

```
;; 如果current-color是'yellow时我们计算(next),
;; 那么其后current-color就是'red

;; 如果current-color是'red时我们计算(next),
;; 那么其后current-color就是'green
```

与此不同，状态变量 *address-book* 可以代表无限个值，所以不可能举出所有的例子。但是，举出一些例子还是非常重要的，因为例子可以使得以后开发函数的主体更简单：

```
;; 如果address-book是empty
;; 计算(add-to-address-book 'Adam 1),

;; 那么其后address-book就是(list (list 'Adam 1))。
;; 如果address-book是(list (list 'Eve 2))
;; 计算(add-to-address-book 'Adam 1),
;; 那么其后address-book就是(list (list 'Adam 1) (list 'Eve 2))。

;; 如果address-book是(list E-1 ... E-2)
;; 计算(add-to-address-book 'Adam 1),
;; 那么其后address-book就是(list (list 'Adam 1) E-1 ... E-2)。
```

在例子中，我们用到了表示时间的文字“其后”，这并不奇怪，毕竟，赋值就是要强调时间的概念。

警告：状态变量永远不会是某个函数的参数。

模板：改变状态的函数的模板与普通函数的模板很相似，只是其主体中应包含set!表达式，用来修改状态变量：

```
(define (fun-for-state-change x y z)
  (set! a-state-variable ...))
```

计算 *a-state-variable* 下一个值的任务可以交给一个读入 *x*、*y* 和 *z* 的辅助函数处理。我们的两个例子就是这样的。

有时候，按照函数输入的定义，我们还需要使用选择器和 cond 表达式。这里再一次考虑 *next*，其输入的数据定义暗示我们使用一个 cond 表达式：

```
(define (next)
  (cond
    [(symbol=? 'green current-color) (set! current-color ...)]
    [(symbol=? 'yellow current-color) (set! current-color ...)]
    [(symbol=? 'red current-color) (set! current-color ...)]))
```

对于这个简单的例子，我们可以使用任意一种设计诀窍设计正确的程序。

主体：与以往一样，开发完整的函数需要对例子有深入的理解，理解它们是怎样被计算的，还要理解函数的模板。对于有效果的函数来说，最需要注意的步骤就是 set!表达式的执行。在某些情况下，赋值的右部只是原始操作、函数参数和状态变量（或者是几个状态变量）。对于其他情况，最好设计一个（没有效果的）辅助函数，读入状态变量当前的值以及函数参数，返回新的状态变量的值。

函数 *add-to-address-book* 就是第一种情况的一个例子。set!表达式的右部仅仅由 *address-book*、*cons* 和 list 组成。而交通信号灯是两种方法都可以选用的一个例子，下面是基于模板的定义：

```
(define (next)
  (cond
    [(symbol=? 'green current-color) (set! current-color 'yellow)]
    [(symbol=? 'yellow current-color) (set! current-color 'red)]
```

```
    [(symbol=? 'red current-color) (set! current-color 'green)]))
```

要写出一个基于辅助函数的定义也很简单：

```
(define (next)
  (set! current-color (next-color current-color)))
```

测试：以前，我们已经测试过许多函数，测试的方法是把例子转化成布尔值表达式，再把它们放到 Definitions 窗口的底部。对于有效果的函数，我们可使用相同的方法，但是证实函数对于某种状态变量有着预期的效果是一个复杂的任务。

有两种方法可以用来测试有效果的函数。第一种方法，可以把状态变量设置为所需的状态，再调用函数，然后检查函数的结果和效果是不是预期的值。*next* 函数就很适合使用这种方法。我们用三个例子完整地描述它的行为，这三个例子都可以被转化成 begin 表达式，用来进行测试。下面就是其中的一个例子：

```
(begin (set! current-color 'green)
       (next)
       (symbol=? current-color 'yellow))
```

其中第一行把状态变量 *current-color* 设置为所需的颜色，第二行计算(*next*)，第三行检查其效果是否正确。我们也可以对 *add-to-address-book* 函数进行类似的测试：

```
(begin (set! address-book empty)
       (add-to-address-book 'Adam 1)
       (equal? '((Adam 1)) address-book))
```

在这个测试中，我们只检查 Adam 和 1 是否被正确地添加到初始的 empty 表中。

第二种方法是，我们可以在测试前保存某个状态变量的值，再调用改变记忆的函数，然后进行合适的测试。考虑如下的表达式：

```
(local ([define current-value-of address-book])
  (begin
    (add-to-address-book 'Adam 1)
    (equal? (cons (list 'Adam 1) current-value-of) address-book)))
```

在计算开始之时，它把 *current-value-of* 定义为 *address-book* 的值，而在计算结束之时，它检查特定的条目是否被添加到了状态变量的前部，而状态变量的其余部分保持不变。

要对有效果的函数进行测试，特别是进行第二种测试，把测试表达式抽象成一个函数是很效的手段：

```
;; test-for-address-book : symbol number -> boolean
;; 判断 add-to-address-book 是否对 address-book
;; 产生了正确的效果，而且没有多余的效果：
;; 与(add-to-address-book name number)相同
(define (test-for-address-book name number)
  (local ([define current-value-of address-book])
    (begin
      (add-to-address-book name number)
      (equal? (cons (list name number) current-value-of)
              address-book))))
```

使用这个函数，现在可以轻易地对 *add-to-address-book* 进行多次测试，并确保每一次测试它的效果都是正确的：

```
(and(test-for-address-book 'Adam 1)
```

```
(test-for-address-book 'Eve 2)
(test-for-address-book 'Chris 6145384))
```

其中 and 表达式保证测试表达式按照顺序计算，并且全部返回 true。

将来的重用：一旦得到了完整的、经过测试的函数，我们应当记住它们的存在，记住它们计算了什么，记住它们的效果是什么。不过，我们并不需要记住它们是怎样计算的。如果遇到一种情况，需要相同的计算以及相同的效果，那么我们可以重用这个程序，就好像它是一个基本操作一样。警告：在效果出现了之后，要重用一个函数比在代数程序的世界中要困难得多。

图 36.2 和 36.3 总结了我们所用的两个例子：第一个例子的头部被省略了，因为就这个特定的例子来说，有用途和效果的说明就足够了。

```
;; 数据定义：TL-color 是'green、'yellow 或'red 三者之一。

;; 状态变量：
;; current-color：TL-color
;; 保存交通信号灯的当前颜色
(define current-color 'red)

;; 合约：next : -> void

;; 用途：该函数总是返回(void)

;; 效果：改变 current-color，从'green 变为'yellow，
;; 'yellow 变为'red，而'red 变为'green

;; 头部：（在这个特定的例子中）省略

;; 例子：
;; 如果 current-color 是'green 时我们计算(next)，那么它变为'yellow
;; 如果 current-color 是'yellow 时我们计算(next)，那么它变为'red
;; 如果 current-color 是'red 时我们计算(next)，那么它变为'green

;; 模板：改变状态变量所指的数据
;; (define (f)
;; (cond
;; [(symbol=? 'green current-color) (set! current-color ...)]
;; [(symbol=? 'yellow current-color) (set! urrent-color ...)]
;; [(symbol=? 'red current-color) (set! current-color ...)]))

;; 定义：
(define (next)
  (cond
    [(symbol=? 'green current-color) (set! current-color 'yellow)]
    [(symbol=? 'yellow current-color) (set! current-color 'red)]
    [(symbol=? 'red current-color) (set! current-color 'green)]))

;; 测试：
(begin (set! current-color 'green) (next) (symbol=? current-color 'yellow))
(begin (set! current-color 'yellow) (next) (symbol=? current-color 'red))
(begin (set! current-color 'red) (next) (symbol=? current-color 'green))
```

图 36.2 状态变量的设计诀窍：一个完整的例子

```
;; 数据定义：任意长的表：(listof X)，两个元素的表：(list Y Z)

;; 状态变量：
;; address-book : (listof (list symbol number))
;; 保存人名和电话号码的对
(define address-book empty)

;; 合约：add-to-address-book : symbol number -> void

;; 用途：该函数永远返回(void)

;; 效果：把(list name phone)添加到 address-book 的前部

;; 头部：
;; (define (add-to-address-book name phone) ...)

;; 例子：
;; 如果 address-book 是 empty 时我们计算
;; (add-to-address-book 'Adam 1),
;; address-book 变为(list (list 'Adam 1))。
;; 如果 address-book 是(list (list 'Eve 2))时我们计算
;; (add-to-address-book 'Adam 1),
;; address-book 变为(list (list 'Adam 1) (list 'Eve 2))。
;; 如果 address-book 是(list E-1 ... E-2) 时我们计算
;; (add-to-address-book 'Adam 1),
;; address-book 变为(list (list 'Adam 1)   E-1 ... E-2)。

;; 模板：省略

;; 定义：
(define (add-to-address-book name phone)
  (set! address-book (cons (list name phone) address-book)))

;; 测试：
(begin (set! address-book empty)
       (add-to-address-book 'Adam 1)
       (equal? '((Adam 1)) address-book))
```

图 36.3　状态变量的设计诀窍：第二个例子

习题

习题 36.4.1　修改图 36.2 中的交通信号灯程序，使它把当前信号灯的状态绘制到画布上。从增加初始化函数的内容开始。使用第 6.2 节的结果。

习题 36.4.2　修改图 36.3 中的电话簿程序，使它提供一个图形用户界面。从增加初始化函数的内容开始。使用习题 35.4.2 的结果。

使用记忆的例子



设计有记忆的程序需要经验和实践，而经验和实践往往是通过研究例子获得的。在这一章中，我们将学习三个使用记忆的程序的例子。第一个例子说明初始化函数的重要性；第二个例子说明如何设计效果取决于条件的程序；最后一个例子说明效果在递归函数中如何有用。本章后两节是应用。

37.1 状态的初始化

回忆习题 5.1.5 中的猜颜色游戏。一位游戏者为两个方块各挑选一种颜色；我们称这为“目标”。另一位游戏者试图猜测哪种颜色被分配给了哪个方块。第一位游戏者比较猜测与他所选的颜色，给出如下的答案：

1. 如果猜测与方块的颜色完全相同，'perfect!；
2. 如果有一个方块（第一个或第二个）的颜色猜对了，'OneColorAtCorrectPosition；
3. 如果猜对了某一种颜色，'OneColorOccurs；
4. 否则，'NothingCorrect。

第一位游戏者只能给出这四个答案之一，第二位游戏者则要用尽可能少的次数猜对所选的两种颜色。

为了简化这个游戏，可供选择的颜色是有限的，参见图 37.1 的前部。我们的目的是开发一个程序，担当控制游戏的角色。也就是说，我们需要一个程序，选择颜色，并检查另一位游戏者的猜测。

```
;; 常数：

;; 合法的颜色
(define COLORS
  (list 'black 'white 'red 'blue 'green 'gold 'pink 'orange 'purple 'navy))

;; 颜色的数量
(define COL# (length COLORS))

;; 数据定义：
;; color 是 COLORS 中的一个符号。
```

图 37.1 猜颜色

游戏的描述说明程序必须提供两种服务：一种服务设立两个目标的颜色，另一种服务检查（另一位游戏者的）猜测。自然，每一种服务对应一个函数。我们把第一种服务称为 *master*，把第二种服务称为

master-check。下面是基于这两个函数的一段可能的会话：

```
> (master)
> (master-check 'red 'red)
'NothingCorrect
> (master-check 'black 'pink)
'OneColorOccurs
...
> (master)
> (master-check 'red 'red)
'perfect!
```

master 函数不读入任何东西，返回不可见的值；它的效果是初始化两个目标。取决于所选颜色的不同，检查同样的两个猜测，可能返回'perfect!，也可能返回'NothingCorrect。换句话说，*master* 设置记忆，*master-check* 使用该记忆。

现在研究设计诀窍是如何用于开发这个程序的。第一步是要定义状态变量，并指定每一个变量的用途。分析表明，我们需要两个状态变量，每个目标各一个：

```
;; target1, target2 : color
;; 这两个变量代表第一位游戏者所选的两种颜色
(define target1 (first COLORS))
(define target2 (first COLORS))
```

这两个变量都被设为 *COLORS* 的第一个元素，所以它们代表了同一种颜色。

第二步是开发两个状态变量的初始化函数。因为这两个变量差不多，所以一个初始化函数就足够了。事实上，初始化函数就是我们所需要的 *master* 函数：

```
;; master : -> void
;; 效果：把 target1 和 target2 设为 COLORS 中随机选择的元素
(define (master)
  (begin
    (set! target1 (list-ref COLORS (random COL#)))
    (set! target2 (list-ref COLORS (random COL#)))))
```

效果注释解释了 *master* 是如何改变两个状态变量的值，它以一个在 0 和 *COLORS* 的大小之间的随机数为基准，从 *COLORS* 选出一个元素。

最后可以开始定义修改和利用记忆的函数了。正如试验所表明的，在两个目标变量被初始化之后，记忆并没有被修改过；我们只是把它和游戏者所作的猜测比较。我们所需要的其他服务只有一个，就是 *master-check*。该函数使用 *check-color*，即习题 5.1.5 所设计的函数，进行比较。图 37.2 是对此的总结，包括了刚才讨论的变量和函数的定义。

习题

习题 37.1.1　绘制图表，说明 *master* 和 *master-check* 是怎样与记忆交互的。

习题 37.1.2　把 *master* 中重复的表达式抽象为函数 *random-pick*，该函数读入一个表，从表中随机选出一个元素，然后使用这个函数消除 *master* 中重复的表达式。

习题 37.1.3　修改猜颜色的程序，使得它最终的答案不仅仅是'*perfect!*，而是一个表，表中包含两个元素：符号 perfect!以及第二个游戏者所猜测的次数。从修改习题 37.1.1 的图表开始。

习题 37.1.4　修改猜颜色的程序，使得它在游戏者猜出目标颜色之后能自动重新开始游戏。

习题 37.1.5　开发一个类似于教学软件包 master.ss 的图形用户界面。不使用彩色按钮，只对按钮

进行颜色标记。请在消息框中显示当前的选择。

```
;; target1, target2 : color
;; 这两个变量代表第一位游戏者所选的两种颜色
(define target1 (first COLORS))
(define target2 (first COLORS))

;; master : -> void
;; 效果：把 target1 和 target2 设为 COLORS 中随机选择的元素
(define (master)
  (begin
    (set! target1 (list-ref COLORS (random COL#)))
    (set! target2 (list-ref COLORS (random COL#)))))

;; master-check : color color -> symbol
;; 判断在多少个位置上猜对了多少个颜色
;; 这个函数与 check-color，习题 5.1.5 的解不同
(define (master-check guess1 guess2)
  (check-color guess1 guess2 target1 target2))
```

图 37.2 猜颜色（第二部分）

37.2 与用户交互并改变状态

回忆第 6.7 节中的刽子手游戏，该游戏的目的是测试某个人掌握的词汇量。一位游戏者想出一个单词，并画出绞刑架的套索；另一位游戏者试着猜出这个单词，每次猜一个字母。对于每一次错误的猜测，第一位游戏者都画出绞刑图像的一个部分（参见图 6.8）：首先画出头，接着是身体、手臂和腿。不过，如果该单词中包含了第二位游戏者所猜测的字母，第一位游戏者就指出该字母在单词中的位置。如果第二位游戏者猜出了完整的单词，或者第一位游戏者完成了整个线条画，游戏就结束了。

图 37.3 给出了字母、单词和身体部件的定义。具体地说，*PARTS* 不仅指定了要画的身体部件，还表明了它们被绘制的顺序。图中还定义了一个不完整的单词表，这样刽子手程序就可以随机地从中选择一个，供用户猜测。

单词的随机选择应在游戏一开始进行，这说明我们需要一个随机的初始化函数，这类似于前一章中的猜颜色程序，不同的是，刽子手程序必须记住游戏者所进行过的猜测次数，因为总的猜测次数是有限的。在'left-leg 被绘制以后，游戏就结束了。对身体部件的倒计数表明，在程序检查猜测的时候，它不仅要告知游戏者这个猜测所指的单词中的该个字母的位置，如果猜测错了的话，还要告知游戏者他现在损失了哪个身体部件。

用一种数据定义来表示这种思想，该数据定义指定了程序合法的响应类型：

> *response*（响应）是下列四者之一：
> 1. "You won"
> 2. (list "The End" word)
> 3. (list "Good guess!" word)
> 4. (list "Sorry" body-part word)

```
;; 数据分析和定义：

;; letter 是下列符号中的一个：'a ……'z 及'_

;; word 是(listof letter)。

;; body-part 是下列符号中的一个：
(define PARTS '(head body right-arm left-arm right-leg left-leg))

;; 常数：
;; 某些待猜的单词：
(define WORDS
  '((h e l l o)
    (w o r l d)
    (i s)
    (a)
    (s t u p i d)
    (p r o g r a m)
    (a n d)
    (s h o u l d)
    (n e v e r)
    (b e)
    (u s e d)
    (o k a y)
    ...
      ))

;; 我们可以选择的单词的数目
(define WORDS# (length WORDS))
```

图 37.3　刽子手程序的基本部分

其中三种响应是表，这样程序就可以一次提供多条信息。具体来说，第一条响应表明在这次填入猜测字母后，单词被完全猜出，所以游戏者赢得了游戏。第二条响应表明了相反的情况，游戏者没有猜对单词中的字母，从而用完了表中所有的身体部件，所以游戏结束。第三条响应所描述的情况是，游戏者这次的猜测是成功的，表中的第二个元素表明现在游戏者所掌握的单词信息。最后一条响应代表了一次失败的猜测，这时响应的表中包含了三个元素：问候语、游戏者所损失的身体部件以及现在游戏者所掌握的单词信息。

现在，我们可以想象程序中两项服务的角色。第一项服务称为 *hangman*，它选择一个新的单词；第二项服务称为 *hangman-guess*，它读入一个字母，返回四种可能的响应中的一种。这是一段可能的会话：

```
> (hangman)
> (hangman-guess 'a)
(list "Sorry" 'head (list '_ '_ '_ '_ '_))
> (hangman-guess 'i)
(list "Good guess!" (list '_ '_ '_ '_ 'i '_))
> (hangman-guess 's)
(list "Good guess!" (list 's '_ '_ '_ 'i '_))
> (hangman-guess 'i)
(list "Sorry" 'body (list 's '_ '_ '_ 'i '_))
...
```

```
> (hangman)
> (hangman-guess 'a)
"You won"
```

这段会话由两轮刽子手游戏组成，说明了 *hangman-guess* 的返回值依赖于上一次 *hangman* 的使用。另外，第一轮游戏还说明，如果两次把 *hangman-guess* 作用于同一个字母，产生的返回值是不同的。这就说明 *hangman-guess* 不仅使用记忆，还修改记忆，具体来说，当游戏者的猜测是错误时，它对身体部件进行倒计数。

另外，这段对话还说明，当一次猜测不能找出单词中新的知识（字母）的时候，游戏者会失去一个身体部件。考虑第二次猜测：'i。*hangman-guess* 的响应表明，它出现在单词中的倒数第二个位置上。用户在第四次猜测时又一次输入了'i，因为'i 的位置已经被揭示了，所以 *hangman-guess* 找不到任何新的进展。在游戏的非正式描述中，我们并没有谈到这个问题。通过这个例子，我们意识到了这一点不明确之处，并给出了结果。

至此，推理表明需要两种服务以及以下三个状态变量：

1. *chosen-word*，要猜的单词；
2. *status-word*，记录已被猜出的字母；
3. *body-parts-left*，记录游戏者还有多少“假想的”身体部分。

正如它们的名字所示，前两个变量总是 *word*。后一个变量自然的取值是身体部件的表；事实上，这个表永远是 *PARTS* 的一个尾部。

图 37.4 给出了状态变量的定义和它们的用途说明。前两个状态变量，*chosen-word* 和 *status-word*，被设为 *WORDS* 的第一个元素，所以它们代表了同一个单词。第三个变量被设为 *PARTS*，因为这就是全部可用的身体部件的集合。

```
;; chosen-word : word
;; 游戏者要猜的单词
(define chosen-word (first WORDS))

;; status-word : word
;; 代表游戏者猜过和还没有猜过的字母
(define status-word (first WORDS))

;; body-parts-left : (listof body-part)
;; 代表还“可以使用”的身体部件
(define body-parts-left PARTS)

;; hangman : -> void
;; 效果：初始化 chosen-word、status-word 和 body-parts-left
(define (hangman)
  (begin
    (set! chosen-word (list-ref WORDS (random (length WORDS))))
    (set! status-word ...)
    (set! body-parts-left PARTS)))
```

图 37.4 刽子手程序的基本部分（第二部分）

下一步，我们必须开发这些状态变量的初始化函数。与前一章中的例子一样，一个初始化函数就足够了。初始化函数就是 *hangman*，它的用途就是设立程序的记忆。具体来说，该函数选择一个单词作为 *chosen-word*，把 *status-word* 和 *body-parts-left* 置为表示游戏刚开始的值。要把 *body-parts-left* 置为表示游戏刚开始的值很简单，因为 *PARTS* 就是这样的表。设置 *status-word* 就需要进行一些分析了。首先，

status-word 的值必定是一个单词；其次，它所包含的字母数必须与 *chosen-word* 相同；最后，因为游戏者还没有进行过任何的猜测，所以其中每一个字母都是未知的。因此，相应的操作应该是用'_创建一个与 *chosen-word* 一样长的单词。

习题

习题 37.2.1　开发函数 *make-status-word*，该函数读入一个单词，返回一个等长的、只由'_组成的单词。使用这个函数完成图 37.4 中 *hangman* 的定义。

习题 37.2.2　使用 build-list，用一个表达式生成 *status word*。完成图 37.4 中 *hangman* 的定义。

现在可以处理问题中最难的部分了：设计 *hangman-guess*，一个使用并修改记忆的函数。该函数读入一个字母，根据 *status-word*、*chosen-word* 和 *guess* 的当前值，返回某种 *response*。同时，如果游戏者猜对了，这个函数必须影响状态变量 *status-word*。如果猜错了，函数必须缩短表示身体部件的表 *body-parts-left*。下面是相应的合约、用途和效果说明：

```
;; hangman-guess : letter -> response
;; 判断游戏者是赢了、输了还是可以继续玩下去，
;; 如果这一次猜错了，求出要失去的身体部件
;; 效果：

;; （1）如果猜测正确，修改 status-word
;; （2）如果猜测错误，缩短 body-parts-left 一个元素
```

我们已经考虑过 *hangman-guess* 的一个会话。仔细分析这段会话，就可以开发 *hangman-guess* 特定的例子。

对话例子和用途 / 效果表明，*hangman-guess* 的返回值依赖于本次猜测正确与否，如果不正确的话，它还依赖于本次猜测是不是最后一次猜测。根据这几种不同，我们来设计一些例子：

1. 如果 *status-word* 是(list 'b '_ '_ '_)，*chosen-word* 是(list 'b 'a 'l 'l)，那么计算

```
(hangman-guess 'l)
```

返回(list "Good guess!" (list 'b '_ 'l 'l))，并且 *status-word* 变成(list 'b '_ 'l 'l)。

2. 如果 *status-word* 是**(list 'b '_ 'l 'l)**，*chosen-word* 是(list 'b 'a 'l 'l)，那么计算

```
(hangman-guess 'a)
```

返回"You won"。在这种情况下，该计算没有效果。

3. 如果 *status-word* 是(list 'b '_ 'l 'l)，*chosen-word* 是(list 'b 'a 'l 'l)，而且 *body-parts-left* 是(list 'right-leg 'left-leg)，那么计算

```
(hangman-guess 'l)
```

返回(list "Sorry" 'right-leg (list 'b '_ 'l 'l))，并且 *body-parts-left* 变成(list 'left-leg)。

4. 最后，如果 *status-word* 是(list 'b '_ 'l 'l)，*chosen-word* 是(list 'b 'a 'l 'l)，而且 *body-parts-left* 是(list 'left-leg)，那么计算

```
(hangman-guess 'l)
```

返回(list "The End" (list 'b 'a 'l 'l))，并且 *body-parts-left* 变成 empty。

前两个例子描述游戏者猜对字母时所发生的事；后两个例子描述游戏者猜错字母时所发生的事。

这几种情况表明，使用的基本模板应基于可能情况的区分：

```
(define (hangman-guess guess)
  (cond
   [... ;; 猜对了字母：
   (cond
```

```
        [... ;; 猜出了完整的单词
         ...]
        [... ;; 还没有猜出完整的单词
             (begin
               (set! status-word ...)
            ...)])]
       [... ;; 没有猜对字母:
        (begin
          (set! body-parts-left ...)
          ... )]))
```

在这个模板中，set!表达式位于嵌套的cond之内，我们要正确描述哪种条件会产生哪种效果。首先，最外层的条件辨别 *guess* 是不是隐含的单词中（尚未猜测出的）字母；如果不是，函数就修改 *body-parts-left*；其次，如果 *guess* 是隐含的单词中（尚未猜测出的）字母，函数就修改 *stotus-word* 变量，除非游戏已经猜完了整个单词。

到目前为止我们还没有考虑过如何表达这些测试，所以先用注释来表示这些条件。这里我们先来处理这个问题，然后再从完整的模板开始定义函数。第一个丢失的条件是，*guess* 是不是隐含的单词中（尚未猜测出的）字母。这里必须比较 *guess* 和 *chosen-word* 中的字母，通过比较给出新的 *status word*。进行比较的辅助函数是：

```
;;reveal-list : word word letter -> word
;;用 chosen-word、status-word 和
;;guess 计算新的 status word
(define (reveal-list chosen-word status-word guess) ...)
```

幸运的是，我们已经两次讨论过这个辅助函数了（参见第6.7节和习题17.6.2），也知道如何设计它了；图37.5包含了一个合适的定义。使用 *reveal-list*，现在可以写出条件，判断 *guess* 对不对：

(equal? *status-word* (*reveal-list status-word chosen-word guess*))

这个条件使用 equal?来比较 *status-word* 的当前值和 *reveal-list* 计算出的新值。如果这两个表相等，那么 *guess* 就是错误的；否则它就是正确的。

第二个丢失的条件是，*guess* 有没有完成单词的猜测。如果 *guess* 就是 *status-word* 中所有未猜出的字母，那么游戏者已经找到了完整的单词，相应的条件是：

```
(equal? chosen-word (reveal-list status-word chosen-word guess))
```

也就是说，如果 *chosen-word* 与 *reveal-list* 的返回值相等，游戏就可以结束了。

把所有的东西都放到同一个模板中，可得：

```
(define (hangman-guess guess)
  (local ((define new-status (reveal-list status-word chosen-word guess)))
    (cond
      [(equal? new-status status-word)
       (begin
      (set! body-parts-left ...)
     ... )]
       [else
       (cond
        [(equal? new-status chosen-word)
      ...]
        [else
        (begin
```

```
      (set! status-word ...)
      ...)])])))
```

因为 *reveal-list* 的返回值被用到了两次，所以在模板中使用了一个 local 表达式。另外，外层的两个条件子句被交换了，因为(equal? *new-status status-word*)比其否定条件更自然。现在可以转而定义函数了。

因为模板是条件的，所以我们分别开发每一个子句：

1．假设(equal? *new-status status-word*)计算出 true，也就是说，游戏者猜错了。这表明游戏者丢失了其身体的一个假想部件。要描述这样的效果，set!表达式必须修改 *body-parts-left* 的值。具体来说，它必须把该状态变量的值设为其当前值的其余部分：

```
(set! body-parts-left (rest body-parts-left))
```

函数的返回值取决于 *body-parts-left* 的新值。如果新值是 empty，游戏就结束了，相应的返回值就是(list "The End" *chosen-word*)，让游戏者知道所选的这个单词到底是什么。如果 *body-parts-left* 不是 empty，相应的返回值就是(list "Sorry" *???* *status-word*)。这个响应表明这次猜错了，其中的第三个部分是 *status-word* 的当前值，这样游戏者可以知道他已经猜出了些什么。这里的*???*代表了一个问题。要理解这个问题，先来看一下我们已经有了些什么：

```
(begin
  (set! body-parts-left (rest body-parts-left))
  (cond
    [(empty? body-parts-left) (list "The End" chosen-word)]
    [else (list "Sorry" ??? status-word)]))
```

原则上，*???*代表的就是游戏者刚刚在绞刑架上失去了哪一个身体部件。但是，因为 set!已经修改过 *body-parts-left* 了，所以我们不能在这里使用(first *body-parts-left*)。正如第 35.2 节中所说的，当使用 set!编程时，时间的选择很重要。我们可以用一个 local 表达式，在修改状态变量之前命名 *body-parts-left* 的第一个元素，从而解决这个问题。

2．第二种情况要比第一种简单得多，我们要区分两种子情况：

a．如果 *new-status* 等于 *chosen-word*，那么游戏者赢了。此时的响应是"You won"；没有效果。

b．如果两者不相等，那么游戏者猜出了一些东西，程序必须告诉他这一点。另外，函数必须要可以继续被调用；(set! *status-word new-status*) 完成这样的效果。此时的响应由一个鼓励语和新的状态组成。

图 37.5 给出了 *hangman-guess* 完整的定义。

习题

习题 37.2.3　画出图表，说明 *hangman* 是 *hangman-guess* 是怎样用状态变量交互的。

习题 37.2.4　用布尔值表达式表示 *hangman-guess* 的四个例子，如果 *hangman-guess* 是正确的，该布尔值表达式返回 true。对每一种情况，再开发一个例子；把这些新的例子转化为补充的测试。

习题 37.2.5　开发一个类似于教学软件包 hangman.ss 的图形用户界面。把这一章中的函数作为回调函数，连接到该界面。

习题 37.2.6　修改刽子手程序，使它记住所有的猜测。这样，如果游戏者在同一轮游戏中重复猜测某个字母，*hangman-guess* 的响应就是"You have used this guess before"。

习题 37.2.7　考虑如下的 *reveal-list*!的变体

```
;; reveal-list! : letter -> void
;; 效果：基于对 chosen-word、status-word
;; 和游戏者的猜测的比较修改 status-word
(define (reveal-list! cw sw guess)
  (local ((define (reveal-one chosen-letter status-letter)
          (cond
```

```
         [(symbol=? chosen-letter guess) guess]
         [else status-letter])))
    (set! status-word (map reveal-one cw sw))))
```

该函数把状态变量 status-word 的值改为一个从 status-word 原来的值、chosen-word 和猜测中计算出的值。

修改 hangman-guess，使得它能与这个 reveal-list!函数一起正确工作。

```
;; hangman-guess : letter -> response
;; 判断游戏者是赢了、输了还是可以继续玩下去，
;; 如果这一次猜错了，求出要失去的身体部件
;; 效果：（1）如果猜测正确，修改 status-word
;; （2）如果猜测错误，缩短 body-parts-left 一个元素
(define (hangman-guess guess)
  (local ((define new-status (reveal-list chosen-word status-word guess)))
    (cond
      [(equal? new-status status-word)
       (local ((define next-part (first body-parts-left)))
         (begin
           (set! body-parts-left (rest body-parts-left))
           (cond
             [(empty? body-parts-left) (list "The End"chosen-word)]
             [else (list "Sorry" next-part status-word)])))]
      [else
       (cond
         [(equal? new-status chosen-word) "You won"]
         [else
          (begin
            (set! status-word new-status)
            (list "Good guess!" status-word))])])))

;; reveal-list : word word letter -> word
;; 计算新的 status-word
(define (reveal-list chosen-word status-word guess)
  (local ((define (reveal-one chosen-letter status-letter)
            (cond
              [(symbol=? chosen-letter guess) guess]
              [else status-letter])))
    (map reveal-one chosen-word status-word)))
```

图 37.5 刽子手程序基本部分（第三部分）

37.3 在递归中改变状态

影响程序记忆的函数不仅可以处理简单形式的数据，也可以处理任意长的数据。要理解这是怎样做到的，让我们仔细地研究一下刽子手游戏程序中 *reveal-list* 的用途。

正如我们刚才所看到的，这个函数比较 *guess* 与 *chosen-word* 中的每一个字母，如果它们相等，*guess* 可以在单词中正确的位置上揭示出新猜出的字母；否则，*status-word* 中相应的字母表示游戏者已经知道

的东西。

函数 *hangman-guess* 接着比较 *reveal-list* 的返回值和 *status-word* 的原值，判断游戏者的猜测正确与否。另外，如果游戏者猜对了，该返回值还要与 *chosen-word* 进行比较，因为 *guess* 可能完成了整个单词的猜测。显然，这两个比较需要重复计算 *reveal-one*。问题是，*reveal-one* 的返回值对 *reveal-list* 来说是有用的，而在 *hangman-guess* 的条件中，它的每一个比较也是有用的。

使用另外一条记忆记录 *reveal-one* 有没有找出新的单词的状态变量，就可以解决问题的前一部分。我们称这个状态变量为 *new-knowledge*，如果 *reveal-one* 判断出 *guess* 找到了一个在 *chosen-word* 中目前被隐藏着的字母，它就修改该状态变量。*hangman-guess* 函数可以使用 *new-knowledge* 来查明 *reveal-one* 发现了什么。

现在把这种想法转变成新的、系统的定义。第一步，我们需要指定状态变量以及它的含义：

```
;; new-knowledge : boolean
;; 该状态变量代表最近一次 reveal-list 的调用正确与否
(define new-knowledge false)
```

第二步，我们必须考虑初始化这个新的状态变量意味着什么。就我们所知，每次把 *reveal-list* 作用于 *guess* 都要用到该状态变量。在调用开始时，该状态变量应该是 false；如果 *guess* 是有用的，它就应该被改为 true。这表明每次调用 *reveal-list* 时，*new-knowledge* 要被初始化成 false。通过改变 *reveal-list*，使它在开始计算其他任何东西以前设置状态变量，就可以完成初始化。

```
;; reveal-list : word word letter -> word
;; 计算新的 status word
;; 效果：首先把 new-knowledge 设置为 false
(define (reveal-list chosen-word status-word guess)
  (local ((define (reveal-one chosen-letter status-letter) ...))
    (begin
      (set! new-knowledge false)
      (map reveal-one chosen-word status-word))))
```

带下划线的表达式就是关键的修改。local 表达式先定义辅助函数 *reveal-one*，然后计算 **local** 的主体。该主体的第一个步骤就是初始化 *new-knowledge*。

第三步，我们必须开发修改 *new-knowledge* 的程序。这个程序已经存在了：*reveal-list*，所以我们的任务是修改它，使它正确地改变状态变量。用一个修改过的效果说明来描述这种思想：

```
;; reveal-list : word word letter -> word
;; 计算新的 status -word
;; 效果：
;; (1) 先把 new-knowledge 设置为 false
;; (2) 如果 guess 正确，把 new-knowledge 设置为 true
```

这个效果的第一部分对于第二部分来说是必需的；一个有经验的程序员可以在效果说明中省略第一部分。

接下来应该修改函数的例子，说明发生了什么样的效果。这个函数的效果是，通过检查 *guess* 有没有在 *chosen-word* 中出现，计算新的 *status -word*。取决于 *guess* 有没有猜出新的字母，存在两种基本情况：

1. 如果 *status-word* 是(list 'b '_ 'l 'l)，同时 *chosen-word* 是(list 'b 'a 'l 'l)，那么计算

   ```
   (reveal-one chosen-word status-word 'a)
   ```

返回(list 'b 'a 'l 'l)，并且 *new-knowledge* 为 true。

2. 如果 *status-word* 是(list 'b '_ '_ '_)，同时 *chosen-word* 是(list 'b 'a 'l 'l)，那么计算

```
(reveal-one chosen-word status-word 'x)
```

返回(list 'b '_ '_ '_)，并且 *new-knowledge* 为 false。

3. 如果 *status-word* 是(list 'b '_ '_ '_)，同时 *chosen-word* 是(list 'b 'a 'l 'l)，那么计算

```
(reveal-one chosen-word status-word 'l)
```

返回(list 'b '_ 'l 'l)，并且 *new-knowledge* 为 true。

4. 最后，如果 *status-word* 是(list 'b '_ 'l 'l)，同时 *chosen-word* 是(list 'b 'a 'l 'l)，那么计算

```
(reveal-one chosen-word status-word 'l)
```

返回(**list 'b '_ 'l 'l**)，而且 new-knowledge 为 **false**。

前两个例子覆盖了基本的情况；第三个例子说明，如果 *guess* 揭示了单词中某些新位置上的字母，*new-knowledge* 也会变成 true；最后一个例子说明，又一次猜测一个已经被揭示出的字母并不能增加对未知单词的认识。

既然已经有了一个函数，就可以跳过模板，直接把注意力集中于现有的函数应该如何修改。现有的 *reveal-list* 版本把 *reveal-one* 作用于两个单词，也就是两个字母表。*reveal-one* 函数比较 *guess* 与 *chosen-word* 中的字母，并判断游戏者有没有找到新的知识。因此必须修改这个辅助函数，让它辨认什么时候 *guess* 代表了正确的猜测，从而把 *new-knowledge* 设为 true。

正如现在所定义的，*reveal-one* 仅仅比较 *guess* 和 *chosen-word* 中的字母。如果 *guess* 与 *chosen-letter* 相等，它并不检查游戏者是不是真的做出了正确的猜测。不过，当且仅当在 *status-word* 中相应的字母还是'_时，字母 *guess* 代表新的字母。这表示需要两处修改，如图 37.6 所显示。也就是说，当且仅当(symbol=? *chosen-letter guess*)和(symbol=? *status-letter* '_)都为真的时候，*reveal-one* 修改 *new-knowledge* 的值。

```
;; reveal-list : word word letter -> word
;; 计算新的 status word
;; 效果：如果 guess 正确，把 new-knowledge 设为 true
(define (reveal-list chosen-word status-word guess)
  (local ((define (reveal-one chosen-letter status-letter)
            (cond
              [(and (symbol=? chosen-letter guess)
                    (symbol=? status-letter '_))
               (begin
                 (set! new-knowledge true)
                 guess)]
              [else status-letter])))
    (begin
      (set! new-knowledge false)
      (map reveal-one chosen-word status-word))))
```

图 37.6 reveal-list 函数

总而言之，如果我们想要把某些结果从一个计算传送到远处，就可以使用状态变量。对上述情况来说，函数的界面已在我们的控制之下，要考虑的是如何设计它，使该函数既有返回值，又有效果。要完成这些的结合，正确的实现方法是分别开发每个计算，然后，需要的话，再把它们融合起来。

习题

习题 37.3.1 画出图表，说明 hangman、hangman-guess 和 reveal-list 是怎样通过状态变量交互的。

习题 37.3.2 把三个例子转变成测试，即布尔值表达式，并测试新版本的 *reveal-list*。问对于第三种情况，*reveal-one* 修改了多少次 *new-knowledge*？

习题 37.3.3 修改刽子手程序中的 hangman-guess，利用 reveal-list 通过 new-knowledge 提供额外信

息。

习题 37.3.4　再一次修改刽子手程序，除去 *hangman-guess* 中的第二个 equal?。提示：引入一个状态变量，记录游戏者还不知道的字母的数目。

我们再来研究另一个函数的例子，该函数读入任意长的数据，并修改程序的记忆。这个例子是第 36 章中交通信号灯的自然扩展。我们曾开发过以下两个函数：

```
;; init-traffic-light : -> void
;; 效果：（1）初始化 current-color；（2）绘制交通信号灯
;; next : -> void
;; 效果：（1）改变 current-color，'green 变为'yellow，
;; 'yellow 变为'red，'red 变为'green
;; （2）绘制相应的交通信号灯
```

前一个函数启动过程；有了第二个函数，我们可以通过在 Interactions 窗口中计算(*next*)反复转换信号灯的状态。

一再地键入(*next*)是很麻烦的，所以自然我们就会想到要写一个程序，能够 100 次、1000 次或者 10000 次转换交通灯的状态。换一种说法，我们应当开发一个程序——我们称它为 *switch*——它读入一个自然数，转换信号灯的状态，从一种颜色转为另一种，一直转换那么多次。

这个函数读入一个自然数，在完成了足够多次的信号灯转换后，返回(void)。现在，我们可以立即写出一个读入自然数的函数的基本部分，包括模板：

```
;; switch : N -> void
;; 用途：这个函数不计算任何的东西
;; 效果：n 次转换交通信号灯，保持每种颜色三秒钟
(define (switch n)
  (cond
    [(zero? n) ...]
    [else ... (switch (- n 1)) ...]))
```

这是传统的、结构递归函数的模板。

要构造一个例子也相当简单。如果计算(*switch* 4)，我们希望看到信号灯从'red 变为'yellow，再变为'green，然后又一次变为'red，每一个时期保持三秒钟可见。

以模板为基础，定义完成的函数是非常简单的。我们按情况处理。如果 *n* 是 0，相应的答案是(void)。否则，我们知道

```
(switch (- n 1))
```

会模拟除了一次以外所有需要的转换动作。要完成这另外的一次转换，函数必须使用(*next*)来执行所有的状态改变，画布的改变，还必须等待三秒钟。如果我们把所有的事都放到一个 begin 表达式中，事情就会以正确的顺序发生：

```
(begin (sleep-for-a-while 3)
       (next)
       (switch (- n 1)))
```

图 37.7 的前部给出了 *switch* 的完整定义。

另一种方法是，不断地转换交通信号灯，至少等到某些外部事件打断这个过程为止。在这种情况下，模拟函数并不读入任何的参数，当它被调用后就会永久地运行。下面是可能遇到的最简单的生成递归形式：

```
;; switch-forever : -> void
```

```
;; 效果：不断地转换交通信号灯
;; 保持每种颜色三秒钟
(define (switch-forever)
  ...
  (switch-forever))
```

```
;; switch : N -> void
;; 效果：n 次转换交通信号灯，保持每种颜色三秒钟
;; 结构递归
(define (switch n)
  (cond
    [(= n 0) (void)]
    [else (begin (sleep-for-a-while 3)
                 (next)
                 (switch (- n 1)))]))

;; switch-forever : -> void
;; 效果：不断地转换交通信号灯，保持每种颜色三秒钟
;; 生成递归
(define (switch-forever)
  (begin (sleep-for-a-while 3)
         (next)
         (switch-forever)))
```

图 37.7　转换交通信号灯的两种方法

因为这个程序在任何条件下都不中止，所以模板只包含了一个递归调用。这就保证可以构造出一个无限循环函数。

使用这个模板，我们可以像以前一样定义完成的函数。在递归之前，函数必须先等待一段时间，再用 *next* 转换信号灯。我们可以用一个 begin 表达式实现这些东西，就如图 37.7 后部的定义。

总而言之，在必须开发修改程序记忆的递归函数时，我们选择与现实情况最相配的设计诀窍，依照它来进行设计。特别，如果函数既有用途，又有效果（例如 *reveal-list* 的例子），我们应当先开发纯粹的函数，然后再添加上效果。

习题

习题 37.3.5　在第 30.2 节中，我们讨论了怎样在简单图中搜索路线。简单图的 Scheme 表示法是符号对的表。符号对说明图中节点的指向关系。每一个节点都正好是一个连接的起点，但可能是多个连接的终点，也可能不是任何连接的终点。给定一张简单图中的两个节点，问题是要找出能不能从前一个节点走到后一个节点。

回忆我们第一个试图判断这样的路线存在与否的函数（可参见图 30.4）：

```
;; route-exists? : node node simple-graph -> boolean
;; 判断在 sg 中是否存在一条从 orig 到 dest 的路径
;; 生成递归
(define (route-exists? orig dest sg)
  (cond
    [(symbol=? orig dest) true]
    [else (route-exists? (neighbor orig sg) dest sg)]))
```

该函数检查源节点和目标节点是不是同一个。如果不是，它生成一个新的问题，查找图中源节点的邻居。

如果图中包含了循环，有时 *route-exists?*就不能返回一个答案。第 30.2 节用一个累积器解决了这个问题。还可以用一个状态变量来解决这个问题，该状态变量记录 *route-exists?*在某次特定的尝试中已经当作源节点访问过的节点。适当地修改这个函数。

习题 37.3.6　在第 16.2 节中，我们开发了计算机文件系统的几个简单模型。开发函数 *dir-listing*，该函数读入一个目录，返回一个表，表中包含该目录中所有文件名和子目录。这个函数还把状态变量 *how- many- directories* 设为它在处理过程中遇到的子目录数。

37.4　状态变量的练习

习题 37.4.1　修改习题 9.5.5 中的 *check-guess-for-list*，使得它还记录游戏者已经按了多少次界面上的“Check”按钮。提示：每一次单击按钮，它就会使用相应的回调函数 *check-guess-for-list* 一次。

习题 37.4.2　开发一个程序，管理一个任务队列。这个程序应当至少支持四种服务：

1．*enter*，添加一项任务到队列的尾部；

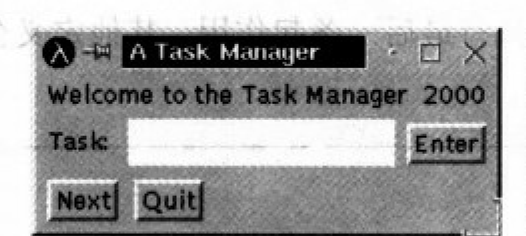
2．*next*，如果有的话，求出队列中下一项任务是什么；

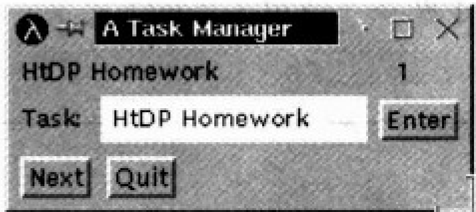
3．*remove*，如果有的话，移去队列中的第一项任务；

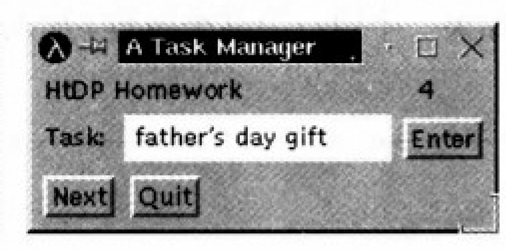
4．*count*，计算队列中元素的总数。

用户可以用 *start-task-manager* 启动任务管理器。

在完成这个程序的开发和测试之后，用 gui.ss 开发一个任务管理器的图形用户界面。该界面应该以一条友好的消息开始，并且总是显示队列中的第一项任务和队列中的元素数：







除非队列是空的，否则单击“Next”按钮应该移去队列中的第一个元素，并显示剩下的队列中的第一个元素是什么。如果队列是空的，单击“Next”按钮应该没有效果。

提示：问候语和年份是两条单独的消息对象。

习题 37.4.3　在第 10.3 节中，我们开发了一个在画布上移动图片的程序。图片是图形的表；这个程序是由绘制、删除和平移图形的函数组成的。其主函数是 *move*（习题 10.3.6）。该函数读入一个图片和自然数 *n*，返回一个新的图片，平移了 *n* 个像素；它还删除原来的图片，绘制新的图片。

开发程序 *drive*。它在画布上绘制一个（固定的）图片，允许游戏者指定移动的像素数，左右移动该图片。修改 *drive*，包含一个燃料的记录，其中每个像素的移动都会消耗固定数量的燃料。

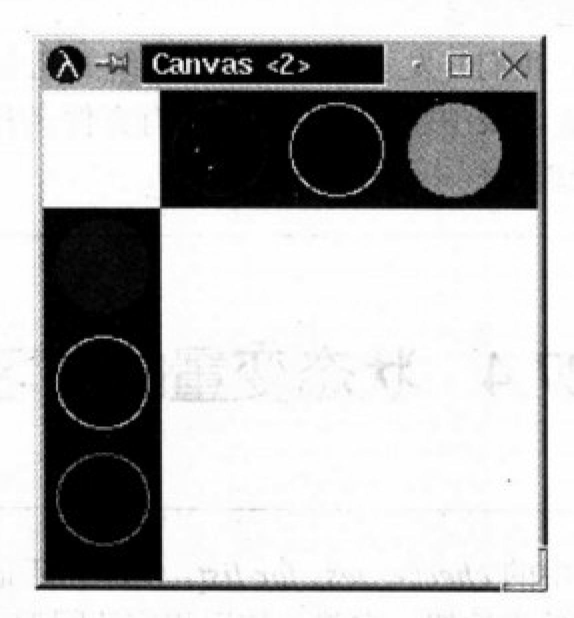
习题 37.4.4　修改控制单个交通信号灯的两个函数，使它们控制一个普通十字路口的两个交通信号灯的状态。每个信号灯都可以是三种状态中的一种：'red、'green 和'yellow。当一个信号灯是'green 或是'yellow 时，另一个灯必须是'red。

回忆这两个单个交通信号灯函数的框架：

```
;; init-traffic-light : -> void
;; 效果：（1）初始化 current-color；（2）绘制交通信号灯
;; next : -> void
```

```
;; 效果：（1）改变current-color，'green变为'yellow，
;; 'yellow变为'red，'red变为'green
;; （2）绘制相应的交通信号灯
```

先修改函数的基本部分。在开发和测试程序时，开发如下图形显示：



使用 init-traffic-light 和 next 函数执行显示，保留其他函数。

习题 37.4.5 在第 14.4 节和第 17.7 节中，我们开发了一个 Scheme 求值程序。一个典型的 Scheme 实现还应当提供一个交互式的用户界面。在 DrScheme 中，Interactions 窗口担任这个角色。

一个交互式系统向读者提示定义和表达式，计算并返回可能的结果。定义被添加到一个知识库中；为了确定这种添加，交互式系统可能会返回一个值，比如 true。表达式使用知识库中相关的定义计算。第 17.7 节中的函数 *interpret-with-defs* 担任这个角色。

开发一个关于 *interpret-with-defs* 的交互系统，该系统至少提供两项服务：

1. *add-definition*，它把某个函数定义（的表示法）添加到系统的知识库中；
2. *evaluate*，它读入某个表达式（的表示法），使用当前知识库中相关的定义计算该表达式。

如果一个用户为某个函数 *f* 加入了两条（或更多条的）定义，只有最后一条起作用，其他定义会被忽略。

37.5 补充练习：探险

早期的电脑游戏要游戏者在危险的迷宫和洞穴中找路。游戏者从一个洞穴走到另一个洞穴，寻找财宝，遭遇各种文明，进行战斗，寻找爱情，获得能量，最终到达天国。这一节，我们使用递归的程序设计方法，设计这样游戏的一个基本部分。

我们的旅程从一个最令人恐惧的地方校园——开始。一个校园由许多建筑组成，某些建筑要比其他的更危险。每个建筑都有名字，并与其他的一些建筑相连。

游戏者总是在某一个建筑物之内。我们把这个建筑物称为当前位置。要了解有关这个位置的更多信息，游戏者可以要求得到该建筑的照片，以及相邻建筑物的表。游戏者还可以发出一个 *go* 命令，移动到一个相邻的建筑物中。

习题

习题 37.5.1 给出建筑的结构体和数据定义。在该结构体中包含一个照片字段。

校园是建筑的表。定义一个简单的校园。图 37.8 是一个例子。

习题 37.5.2　开发一个程序，允许游戏者穿越习题 37.5.1 中校园的例子。该程序应当支持至少三种服务：

1．*show-me*，它返回当前位置的照片：参见图 37.9；

2．*where-to-go*，它返回相连的建筑的表；

3．*go*，它改变游戏者的当前位置。

如果游戏者发出命令(*go s*)，而 *s* 并不与当前位置相连，函数必须报告一个错误消息。在必要的时候，或者按照自己的期望，开发其他函数。

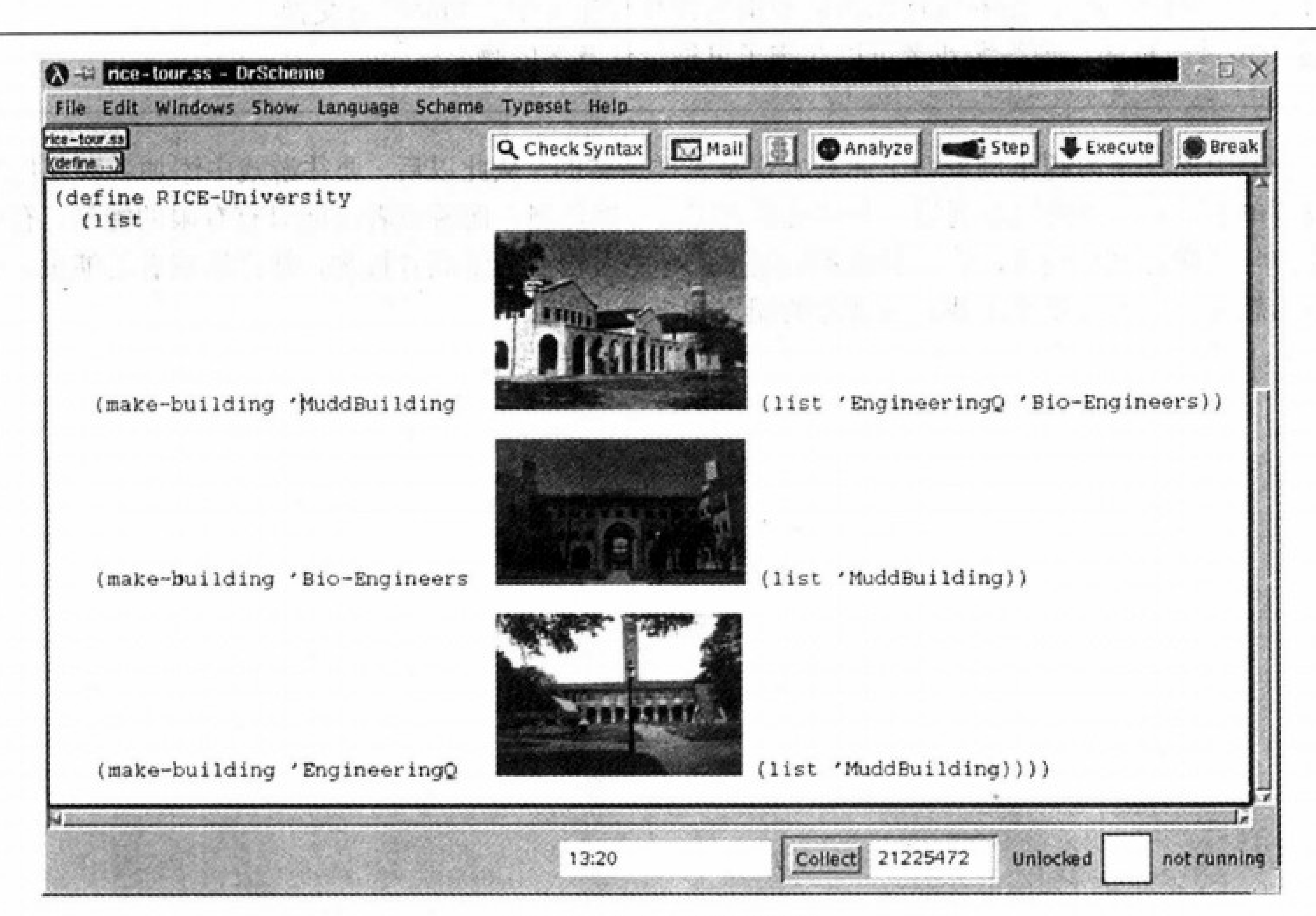


图 37.8　赖斯大学的旅程图



图 37.9　参加旅行

在早期迷宫游戏中，游戏者们还可以在不同的地方收集物品、进行交易。具体来说，游戏者有一个用来装东西的袋子，而每一个地方都包含了一些物品，游戏者可以用自己袋中的东西交换这些物品。

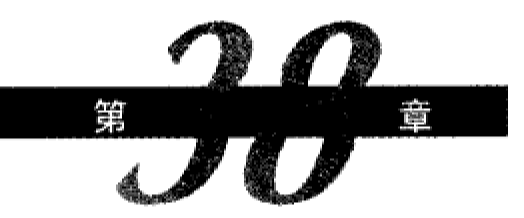
习题

习题 37.5.3　修改旅行程序，使每一个建筑物都包含一样东西。另外，游戏者应该有一个袋子，可以容纳（最多）一样东西。在任何一个位置，如果袋子是空的，游戏者可以捡起一样东西，或者如果袋子里已经有东西了，游戏者可以将袋子里的物品与建筑中的物品进行交换。

进一步修改程序，使得游戏者可以在袋子里携带任意多的物品。

这一节中的三个习题举例说明了迷宫游戏是怎样运转的。从此以后，要往游戏中添加各种不同的东西就很简单了。从一个建筑走到另一个建筑需要用去一些能量，而游戏者可能只有有限的能量。有些生物可能会与游戏者进行战斗，消耗游戏者的能量；有些人物会与游戏者接吻，补充游戏者的能量。充分发挥你的想象力，扩展这个游戏，并请你的朋友们来参加。

最终的语法和语义



介绍过 set!表达式和 begin 表达式之后，我们的讨论就已经遍及了整个 Scheme 语言的大部分。在 DrScheme 中，这个部分被称作 Advanced Student Scheme。考虑到 set!的复杂性，这里该是我们对比第 8 章，总结程序设计语言的地方了。遵照第 8 章，我们讨论词汇、语法和 Advanced Student Scheme 的含义。最后一节解释在 Advanced Student Scheme 中可能遇到的错误类型。

38.1 Advanced Scheme 的词汇

任何语言的基础都是词汇。在 Beginning Student Scheme 中，我们区分四种单词：变量、常数、基本函数和关键字。这种分类忽略了括号，但我们知道每一个复合短语都是由一对括号环绕的，而每一个原子短语都代表了它自己。

Advanced Student Scheme 并不违反这种基本的分类，但是它包含了四个新的关键字：local、lambda、set!和 begin。前两个关键字对组织和抽象程序来说非常重要；后两个关键字对效果的计算来说非常重要。尽管如此，关键字本质上并没有意义。它们只是路标，告诉我们前方是什么，这样我们就可以确定自己的方向。是文法和语言的含义解释了关键字的作用。

38.2 Advanced Scheme 的文法

尽管 Scheme 是一种完全成熟的语言，它的能力与其他程序设计语言一样强大，但是它的设计者仍然保持了其文法的简单。Advanced Student Scheme 的文法包括了大部分的 Scheme 文法，它只比 Beginning Student Scheme 的文法长一点点。

图 38.1 给出了 Advanced Student Scheme 语言的基本文法，它是 Intermediate Student Scheme 的扩展，使用了三种新的表达式：lambda 表达式、set!表达式和 begin 表达式。local 表达式的说明引用了定义类别，而定义类别又引用了表达式类别。lambda 表达式由包含在一个括号中的一连串变量和一个表达式组成。类似地，set!表达式由一个变量和一个表达式组成。最后，begin 表达式只是一连串表达式，由关键字 begin 开始，从而区别于调用。

由于函数是值，Advanced Student Scheme 还在一个方面比 Beginning Student Scheme 简单。具体来说，它把基本操作与函数调用合并到了一行中。新的这一行说明，调用现在是由括号环绕的一连串表达式。由于把基本操作包含到了表达式之中，

(+ 1 2)

仍然是一个表达式。毕竟，现在+是一个表达式，1 和 2 也是表达式。用 define 定义的函数也与此类似：

(*f* 1 2)

其中第一个表达式是一个变量，后两个表达式是数。所以这个调用是一个合法的表达式。

```
<def>  =  (define (<var> <var> ...<var>) <exp>)
         |(define <var> <exp>)
         |(define-struct <var0> (<var-1> ... <var-n>))

<exp>  =  <var>
         |<con>
         |<prm>
         |(<exp> <exp> ...<exp>)
         |(cond (<exp> <exp>) ...(<exp> <exp>))
         |(cond (<exp> <exp>) ...(else <exp>))
         |(local (<def> ...<def>) <exp>)
         |(lambda (<var> ...<var>) <exp>)
         |(set! <var> <exp>)
         |(begin <exp> ...<exp>)
```

图 38.1 advanced student Scheme:核心文法

不幸的是，一种语言的文法只能说明合法语句的框架轮廓，它不能表示需要短语语境知识的约束，而 Advanced Student Scheme 需要一些这样的约束：

1. 在 lambda 表达式中，一个变量不能两次出现在参数序列中；
2. 在 local 表达式中，同一个序列中的（多个）定义不能引入同样的变量；
3. set!表达式必须出现在引入 set!表达式左部的 define 辖域内。

另外，关键字不能被当作变量的约束仍适用。

考虑如下的定义，它使用了新的约束：

```
(define switch
  (local ((define-struct hide (it))
          (define state (make-hide 1)))
    (lambda ()
      (begin
     (set! state (make-hide (- 1 (hide-it state))))
     state))))
```

定义引入了变量 *switch*，定义的右部是一个 local 表达式。这个表达式又定义了结构体 *hide* 与变量 *state*，其中 *state* 代表了 *hide* 的一个实例。local 表达式的主体是一个 lambda 表达式，其参数序列为空。这个函数的主体由一个 begin 表达式组成，该 begin 表达式含有两个表达式：一个 set!表达式和一个仅仅由变量 *state* 组成的表达式。

在这个程序中，所有的表达式都满足必需的约束。首先，local 表达式引入了四个不同的名称：make-hide、hide?、hide-it 和 *state*；其次，lambda 表达式的参数表是空的，所以其中不可能有冲突。最后，set!表达式的变量是由 local 局部定义的变量 *state*，所以也是合法的。

习题

习题 38.2.1 判断下列表达式在语法上是不是合法的程序：

```
1. (define (f x)
     (begin
       (set! y x)
```

```
      x))
  2. (define (f x)
      (begin
        (set! f x)
        x))
  3. (local ((define-struct hide (it))
             (define make-hide 10))
       (hide? 10))
  4. (local ((define-struct loc (con))
             (define loc 10))
       (loc? 10))
  5. (define f
       (lambda (x y x)
         (* x y z)))

     (define z 3.14)
```

解释为什么合法或为何不合法。

38.3 Advanced Scheme 的含义

在我们第一次使用 Advanced Student Scheme 时，是因为想要把函数当作普通的值来处理。计算的规则几乎没有改变，我们只是允许表达式出现在调用的第一个位置之上，从而像值一样处理函数。

set!表达式对语言的扩充需要另外一种规则。现在，联系变量与值的定义可以在计算中改变，但到目前为止，所使用的涉及状态变量定义改变的规则是非正式的，也是不精确的，所以我们需要一个精确的描述来刻划 set!是如何改变语言的含义的。

前面我们是这样确定程序含义的：程序由定义集合和一个表达式两个部分组成，其目标是计算这个表达式，也就是说确定该表达式的值[1]。在 Beginning Student Scheme 中，值的集合包括了所有的常数和表。只有一种表有着精确的表示法：空表。所有其他的表都是一系列 cons 组成的表。

对一个表达式的计算是由一系列步骤组成的。在每一步中，我们使用算术与代数规则来简化一个子表达式，这将返回另一个表达式。我们还说，把第一个表达式重写为第二个，如果后一个表达式是一个值，就结束了对表达式的计算。

把 set!表达式引入到程序设计语言之后，需要对这个过程进行一些调整和扩展：

1. 除了重写表达式以外，我们还必须重写定义。更精确地说，计算的每一步都会改变表达式，也可能会改变某个状态变量的定义。要使这些效果尽可能明显，计算的每个阶段都要显示出状态变量的定义以及当前的表达式。

2. 另外，算术与代数规则不再无论何时何地都适用，取而代之的是，如果要 进行计算，必须判断必须计算的子表达式，但这个规则仍保留了选择。例如在重写

```
(+ (* 3 3) (* 4 4))
```

这样的表达式时，可以选择先计算(* 3 3)后计算(* 4 4)，也可以反过来。幸运的是，对于这种简单的表达式，选择并不会影响最终的结果，所以并不需要提供一条完全明确的规则。不过，一般来说，我们按照从左到右、从上到下的顺序重写表达式。在计算的每一个过程，我们最好从把下一步要计算的子表

[1] 如果定义的右部不是值，我们还将它们计算为值，但是这里我们可以安全地忽略这个小问题

达式画上下划线开始。

3. 假设带下划线的子表达式是一个 set!表达式。依据 set!表达式的约束，我们知道其左部的子表达式已经有了 define。也就是说，我们面对如下的情况：

```
    (define x aValue)
    ...
    ... (set! x anotherValue) ...
  = (define x anotherValue)
    ...
    ... (void) ...
```

这个等式表明，程序在两方面有所改变：第一，变量的定义被修改了；第二，带下划线的 set!表达式被替换成了不可见的值(void)。

4. 下一个改变是关于把表达式中的变量替换成它们定义中的值。到现在为止（引入 set!表达式之前），我们可以在认为必要或方便的时候，把一个变量替换为它的值。实际上，我们只是把变量看作其值的一个简写。语言中有了 set!表达式，这一点就不可行了。毕竟，set!表达式的计算会更改状态变量的定义，如果在错误的时候替换某个变量的值，就会得到错误的值。

假设带下划线的表达式是一个（状态）变量，在变量被替换成它当前的定义值之前，我们不能任意推进计算。这表明了如下修改后的变量计算规则：

```
    (define x aValue)
    ...
    ... x ...
  = (define x aValue)
    ...
    ... aValue ...
```

简而言之，只在需要某个状态变量的值的时候，我们才用该状态变量的定义替换它。

5. 最后，我们还需要 begin 表达式的规则。最简单的一种说法是，如果第一个子表达式是一个值的话就丢弃它：

```
    (begin v exp-1 ... exp-n)
  = (begin exp-1 ... exp-n)
```

这意味着还需要一条规则，用来完全丢弃 begin：

```
    (begin exp)
  = exp
```

另外，为了方便，我们可使用一条一次丢弃多个值的规则：

```
    (begin v-1 ... v-m exp-1 ... exp-n)
  = (begin exp-1 ... exp-n)
```

虽然比起 Beginning Student Scheme 来，这些规则更为复杂，但它们还算易于处理。

我们来考虑一些例子。第一个例子说明子表达式计算顺序的不同如何造成结果的不同：

```
  (define x 5)
  (+ (begin (set! x 11) x) x)

= (define x 11)
  (+ (begin (void) x) x)

= ...
```

```
= (define x 11)
  (+ 11 x)

= (define x 11)
  (+ 11 11)
```

该程序由一个定义和一个加法组成，其中加法是需要计算的。加法的一个参数是一个改变 *x* 的 set!表达式，另一个参数正好是 *x*。从左向右计算加法的子表达式，变量的改变就会在把第二个子表达式替换成其值之前发生。结果，计算的结果就是 22。如果我们从右向左计算加法，返回值会变成 16。要避免这种问题，我们要使用固定的计算顺序，当不涉及状态变量的时候，可以自由一些。

第二个例子说明，在 local 表达式中的一个 set!表达式是怎样改变一个外层定义的：

```
(define (make-counter x0)
   (local ((define counter x0)
      (define (increment)
        (begin
        (set! counter (+ counter 1))
        counter)))
     increment))
  ((make-counter 0))
```

这个程序也是由一个单独的定义和一个需要计算的表达式组成。不过，这里的后者是一个嵌套调用。内层的调用用下划线标出，因为我们必须先计算它的值，从而推进整个计算。下面是一些计算步骤：

```
=(define (make-counter x0)
   (local ((define counter x0)
      (define (increment)
        (begin
        (set! counter (+ counter 1))
        counter)))
     increment))
  ((local ((define counter 0)
      (define (increment)
        (begin
          (set! counter (+ counter 1))
            counter)))
    increment))
=(define (make-counter x0)
   (local ((define counter x0)
         (define (increment)
         (begin
        (set! counter (+ counter 1))
        counter)))
     increment))
  (define counter1 0)
  (define (increment1)
   (begin
    (set! counter1 (+ counter1 1))
    counter1))
```

```
  (increment1)
```

对 local 表达式的计算创建了额外的外层表达式。其中一个表达式引入一个状态变量；其他的表达式定义函数。

计算的第二部分确定(*increment1*)完成了什么：

```
  (define counter1 0)
  (increment1)
=(define counter1 0)
  (begin
    (set! counter1 (+ counter1 1))
    counter1)
=(define counter1 0)
  (begin
    (set! counter1 (+ 0 1))
    counter1)
=(define counter1 0)
  (begin
    (set! counter1 1)
    counter1)
=(define counter1 1)
  (begin
    (void)
    counter1)
=(define counter1 1)
  1
```

在计算中，我们两次把 *counter1* 替换成它的值。第一次，在第二步中，把 *counter1* 替换为 0，即它那时的值；第二次，在最后一步中，把 *counter1* 替换为 1，也就是它的新值。

习题

习题 38.3.1 用下划线标出下列表达式中下一步必须要计算的子表达式：

```
1. (define x 11)
    (begin
      (set! x (* x x))
      x)
2. (define x 11)
    (begin
      (set! x
     (cond
       [(zero? 0) 22]
       [else (/ 1 x)]))
      'done)
3. (define (run x)
      (run x))
    (run 10)
4. (define (f x) (* pi x x))
    (define a1 (f 10))
```

```
   (begin
     (set! a1 (- a1 (f 5)))
     'done)
5. (define (f)
     (set! state (- 1 state)))
   (define state 1)
   (f (f (f)))
```

解释为什么这些表达式必须被计算。

习题 38.3.2　验证带下划线的表达式接下来必须被计算：

```
1. (define x 0)
   (define y 1)
   (begin
     (set! x 3)
     (set! y 4)
     (+ (* x x) (* y y)))
2. (define x 0)
   (set! x
     (cond
   [(zero? x) 1]
   [else 0]))
3. (define (f x)
     (cond
   [(zero? x) 1]
   [else 0]))
   (begin
     (set! f 11)
     f)
```

重写这三个程序，给出下一步的状态。

习题 38.3.3　计算下列程序：

```
1. (define x 0)
   (define (bump delta)
     (begin
   (set! x (+ x delta))
   x))
   (+ (bump 2) (bump 3))
2. (define x 10)
   (set! x (cond
          [(zeor? x) 13]
          [else (/ 1 x)]))
3. (define (make-box x)
     (local ((define contents x)
          (define (new y)
        (set! contents y))
            (define (peek)
        contents))
   (list new peek)))
```

```
(define B (make-box 55))
(define C (make-box 'a))

(begin
  ((first B) 33)
  ((second C)))
```

在每一步中，用下划线标出下一步必须计算的子表达式。给出涉及 local 表达式或 set!表达式的步骤。

原则上，我们可以使用刚才讨论的这些规则进行计算。这些规则覆盖了普通的情况，解释了我们所遇到的程序的行为。不过，它们并没有解释，当赋值的左部引用了一个 define 定义的函数时，赋值是怎样工作的。考虑如下的例子，其中规则仍然适用：

```
  (define (f x) x)

  (begin
    (set! f 10)
f)

=(define f 10)

  (begin
    (void)
    f)
```

这里 f 是一个状态变量。set!表达式改变了定义，使得 f 代表一个数。计算的下一步把 f 的出现替换为 10。

在一般的情况下，赋值会把一个函数定义替换成另一个函数定义。观察下面的程序：

```
(define (f x) x)
(define g f)
(+ (begin (set! f (lambda (x) 22)) 5) (g 1))
```

带下划线的 **set!** 表达式的目的是修改 f 的定义，使得它变为一个返回 22 的函数。但是，起初 g 代表了 f。既然 f 是一个函数的名字，我们可以把(define g f)看作一个值的定义。问题是，我们当前的规则改变了 f 的定义，从而改变了 g 的定义，因为 g 代表了 f:

```
=(define f (lambda (x) 22))
 (define g f)
 (+ (begin (void) 5) (g 1))

=(define f (lambda (x) 22))
 (define g f)
 (+ 5 (g 1))

=(define f (lambda (x) 22))
 (define g f)
 (+ 5 22)
```

然而，Scheme 并不是这样运作的。一个 set!表达式一次只能改变一个定义。这里它改变了两个定义：

f 的和 *g* 的，*f* 是按照我们的目的改变的，*g* 是通过从 *g* 到 *f* 间接实现的。简而言之，我们的规则并没有解释所有包含 set!表达式的程序的行为；如果想要完全了解 Scheme，则需要更好的规则。

这个问题涉及到函数的定义，这提示我们再一次观察函数的表示法与函数的定义。到目前为止，我们把函数的名称当作值来使用。正如我们所看到的，这种选择可能会在状态变量的情况中导致问题。解决的方法是使用一种具体的函数表示法。幸运的是，我们在 Scheme 中已经有了这样一种东西：lambda 表达式。我们重写函数的定义，使它们变为值的定义，其右部为一个 lambda 表达式：

```
  (define (f x) x)

=(define f (lambda (x) x))
```

即使递归定义也可使用这种方法进行计算：

```
  (define (g x)
    (cond
      [(zero? x) 1]
      [else (g (sub1 x))]))
=(define g
    (lambda (x)
      (cond
     [(zero? x) 1]
     [else (g (sub1 x))])))
```

所有其他的规则，包括用把变量替换成它们的值，都保持不变。

```
<vdf>   =   (define <var> <val>)
            | (define-struct <var> (<var> ...<var>))
<val>   =   <con> | <lst> | <prm> | <fun> | <void>

<lst>   =   empty | (cons <val> <lst>)

<fun>   =   (lambda (<var> ...<var>) <exp>)
```

图 38.2　Advanced Student Scheme：值

图 38.2 列出了值的集合和值定义的集合，值的集合是表达式集合的一个子集，值定义的集合是定义集合的一个子集。使用这些定义和修改后的规则，再一次观察上述例子：

```
  (define (f x) x)
  (define g f)
  (+ (begin (set! f (lambda (x) 22)) 5) (g 1))

=(define f (lambda (x) x))
  (define g f)
  (+ (begin (set! f (lambda (x) 22)) 5) (g 1))

=(define f (lambda (x) x))
  (define g (lambda (x) x))
  (+ (begin (set! f (lambda (x) 22)) 5) (g 1))

=(define f (lambda (x) 22))
  (define g (lambda (x) x))
```

```
(+ (begin (void) 5) (g 1))

=(define f (lambda (x) 22))
 (define g (lambda (x) x))
 (+ 5 (g 1))

= (define f (lambda (x) 22))
 (define g (lambda (x) x))
 (+ 5 1)
```

关键的区别是，g 的定义直接与表示函数的变量相连，而不是和函数的名称相连。

下面的程序通过一个极端的例子说明作用于函数的 set!表达式的效果：

```
(define (f x)
 (cond
   [(zero? x) 'done]
   [else (f (sub1 x))]))
(define g f)
(begin
 (set! f (lambda (x) 'ouch))
 (symbol=? (g 1) 'ouch))
```

函数 f 是关于自然数的递归函数，它总是返回'done。一开始，g 被定义为 f。最后的 begin 表达式先修改 f，再使用 g。

首先，必须按照修改后的规则重写函数的定义：

```
=(define f
   (lambda (x)
     (cond
    [(zero? x) 'done]
    [else (f (sub1 x))])))
 (define g f)
 (begin
   (set! f (lambda (x) 'ouch))
(symbol=? (g 1) 'ouch))

=(define f
   (lambda (x)
     (cond
    [(zero? x) 'done]
    [else (f (sub1 x))])))
 (define g
   (lambda (x)
     (cond
    [(zero? x) 'done]
    [else (f (sub1 x))])))
 (begin
  (set! f (lambda (x) 'ouch))
  (set! f (lambda (x) 'ouch))
  (symbol=? (g 1) 'ouch))
```

重写 f 的定义还是简单的。主要的改变是关于 g 的定义。它现在不再包含 f，而是包含了 f 当前所代表的值的副本。这个值中包含了一处对 f 的引用，但这并非不寻常的。

接下来，**set!**表达式修改 f 的定义：

```
  ...
=(define f
   (lambda (x)
'ouch))

  (define g
   (lambda (x)
     (cond
   [(zero? x) 'done]
   [else (f (sub1 x))])))

  (begin
   (void)
   (symbol=? (g 1) 'ouch))
```

不过，没有其他的定义受到影响。特别是，g 的定义保持不变，虽然 g 值中的 f 现在引用了一个新的值。但是我们以前看到过这种现象。接下来的两个步骤遵循第 8 章中的基本规则：

```
  ...
=(define f
   (lambda (x)
'ouch))

  (define g
   (lambda (x)
     (cond
   [(zero? x) 'done]
   [else (f (sub1 x))])))

  (begin
   (void)
(symbol=? (f 0) 'ouch))

= (define f
   (lambda (x)
'ouch))

  (define g
   (lambda (x)
     (cond
   [(zero? x) 'done]
   [else (f (sub1 x))])))

 (begin
  (void)
```



```
(symbol=? 'ouch 'ouch))
```

也就是说，对 *g* 的调用最终把 *f* 作用于 0，其返回'ouch。因此最终的返回值是 true。

习题

习题 38.3.4 验证如下程序的计算结果是 true：

```
(define (make-box x)
  (local ((define contents x)
          (define (new y) (set! contents y))
          (define (peek) contents))
(list new peek)))

(define B (make-box 55))

(define C B)

(and
  (begin
   ((first B) 33)
   true)
  (=(second C) 33)
  (begin
   (set! B (make-box 44))
   (=(second C) 33)))
```

在每一个步骤中，用下划线标出下一步要计算的子表达式，给出涉及 local 表达式和 set!表达式的值。

当决定重写函数定义，使得它的右部总是 lambda 表达式时，我们会遇到一个关于函数调用规则的困难，该规则假设函数定义都仿照 Beginning Student Scheme 的式样。更具体地说，如果定义的环境中包含了这样一个定义：

```
(define f (lambda (x y) (+ x y)))
```

而表达式是：

```
(* (f 1 2) 5)
```

那么计算的下一步是：

```
(* (+ 1 2) 5)
```

不过，在其他情况下，我们仅把变量替换成它在定义中的值。如果遵照这个规则，我们会把

```
(* (f 1 2) 5)
```

重写为

```
(* ((lambda (x y) (+ x y))
     1 2)
   5)
```

初步探测到这里就结束了，因为没有处理这种调用的规则。

我们可以用一条新的规则使这两种思想变得一致，这条新规则可以从下述表达式得出：

```
((lambda (x-1 ... x-n) exp)
 v-1 ... v-n)
```

```
=exp  其中所有的 x-1 ... x-n 都被替换成 v-1 ... v-n
```

该规则的作用是替代代数中的函数应用。按照传统，这条规则被称为 β_v 公理。

***β* 与 *λ* 演算：** 最初的 β 公理是由阿朗索・丘奇在 20 世纪 20 年代后期提出的，形式如下：

```
((lambda (x) exp)
 exp-1)
=exp  其中的 x 被替换成 exp-1
```

它并不限制函数调用的参数一定要是值。丘奇和其他的逻辑学家[1]所感兴趣的是探索计算的原理，计算可以完成什么，以及计算不能完成什么。他们证明了，β 公理加上 Scheme 的一个小型子语言，也就是，

<exp> = *<var>* | (**lambda** (*<var>*) *<exp>*) | (*<exp>* *<exp>*)

就足够定义出所有可计算的处理自然数的（模拟）函数。不能用这种语言表达的函数就是不可计算的。

这种语言与 β 公理后来就成为了大家所知的 λ 演算。杰拉尔德・苏塞曼和盖伊・L・斯蒂尔后来在 λ 演算的基础上建立了 Scheme。20 世纪 70 年代中期，戈登・普洛特金建议大家，把 β 公理当作一种更好地理解函数调用（像 Scheme 这种程序设计语言中的函数调用）的方法。

38.4 Advanced Scheme 中的错误

扩展语言，把函数当成值，不仅为程序员引入了新的能力，但也引入了新的错误可能。回忆一下，总共有三种类型的错误：语法错误，运行错误（或称语义错误）和逻辑错误。Advanced Student Scheme 把 Beginning Student Scheme 中的一类语法错误转变成了运行错误，还引入了一种新的逻辑错误。

考虑如下的程序：

```
;; how-many-in-list : (listof X) -> N
;; 计算 alist 中包含了多少个元素
(define (how-many-in-list alist)
  (cond
    [empty? (alist)]
    [else (+ (how-many-in-list (rest alist)) 1)]))
```

在 Beginning Student Scheme 或 Intermediate Student Scheme 中，因为 *alist* 是函数的参数，但又被当作一个函数使用，所以 DrScheme 会产生一个语法错误消息。在 Advanced Student Scheme 中，因为函数是值，所以 DrScheme 必须认可这个函数定义在语法上是正确的。不过，当这个函数被作用于 empty 或其他的值（表）时，DrScheme 就会把 empty 作用于空参数，这就是一个运行错误。毕竟，表不是函数。DrScheme 立即产生一条关于试图调用非函数的错误消息，并停止计算。

第二种错误类型是逻辑错误。也就是说，一个包含这种错误的程序并不会产生一个语法或运行错误消息，而是会返回错误的返回值。观察如下的两个定义：

```
(define flip1
  (local ((define state 1))
    (lambda ()
      (begin
     (set! state (- 1 state))
```

```
(define flip2
  (lambda ()
    (local ((define state 1))
      (begin
     (set! state (- 1 state))
```

[1] 逻辑在计算中的重要性与数学在物理中的重要性一样。

```
    state))))                                state))))
```

它们之间的区别是其中有两行顺序不同。一个引入了一个 local 定义，它的主体是计算一个函数；另一个则定义一个函数，它的主体包含了一个 local 表达式。按照我们的规则，左边的定义重写后变为

```
(define state1 1)                    (define flip2
                                       (lambda ()
(define flip1                            (local ((define state 1))
  (lambda ()                               (begin
    (begin                               (set! state (- 1 state))
      (set! state1 (- 1 state1))         state))))
      state1)))
```

右边的定义已经联系起了一个名字与一个函数。

现在来看看这两个函数完全不同的表现。要明白这一点，分别在不同的定义环境下计算表达式：

```
(and(= (flip1) 0)                    (and (= (flip2) 0)
    (= (flip1) 1)                         (= (flip2) 1)
    (= (flip1) 0))                        (= (flip2) 0))
```

下面是左边表达式计算的前四个步骤：

```
  (define state1 1)
  (and(=(flip1) 0)
      (=(flip1) 1)
    (=(flip1) 0))

 =(define state1 1)
  (and(=(begin
        (set! state1 (- 1 state1))
        state1)
      0)
      (=(flip1) 1)
    (=(flip1) 0))

 =(define state1 1)
  (and(=(begin
        (set! state1 0)
        state1)
      0)
      (=(flip1) 1)
    (=(flip1) 0))

 =(define state1 0)
  (and(=(begin
           (void)
            state1)
              0)
      (= (flip1) 1)
    (=(flip1) 0))
```

```
=(define state1 0)
  (and(=0 0)
       (=(flip1) 1)
       (=(flip1) 0))
```

与它相应的定义环境是 *state1* 的定义，可以看到，它在第三步中从 1 变为了 0。从这一点来看，不难验证这个表达式会返回 true，而且最后 *state1* 变为 0。

比较这个计算与右边表达式的前三个计算步骤：

```
  (and(=(flip2) 0)
       (=(flip2) 1)
      (=(flip2) 0))

= (and (=(local ((define state 1))
              (begin
              (set! state (- 1 state))
               state))
                 0)
       (=(flip2) 1)
      (=(flip2) 0))

=(define state1 1)
  (and(=(begin
             (set! state1 (- 1 state1))
             state1)
                0)
       (=(flip2) 1)
      (=(flip2) 0))

=(define state1 0)
  (and(=0 0)
       (= (flip2) 1)
       (=(flip2) 0))
```

这里惟一涉及到的定义是 *flip2* 的定义。表面上，这两个计算互相类似，但是仔细观察表明，第二个定义在关键的方面与第一个不同。它会建立 *state1* 的定义，而第一个计算是由这个定义开始的。

下面是第二个计算的继续：

```
  ...
=(define state1 0)
  (and true
       (=(local ((define state 1))
              (begin
              (set! state (- 1 state))
              state))
               1)
(=(flip2) 0))

=(define state1 0)
  (define state2 1)
```

```
    (and true
         (=(begin
               (set! state2 (- 1 state2))
                state2)
                  1)
 (=(flip2) 0))

=(define state1 0)
  (define state2 0)
  (and true
       (=(begin
            (void)
             state2)
               1)
 (=(flip2) 0))

=(define state1 0)
  (define state2 0)
  (and true
       (=0 1)
       (= (flip2) 0))
```

这表明 *flip2* 在每一次被调用时，都会建立一个新的定义，并且它每次都返回 0。与它的名字相反，它并不在每次调用时翻转 *state* 的值，结果是计算会在生成两个新的外层定义之后停止，并返回 false。

经验教训是，用 local 表达式定义的函数与主体包含一个 local 表达式的函数是不同的。前者保证这个定义只能被该函数所使用，定义在这个函数中存在且仅存在一次。反之，后者在每一次计算函数的主体时建立一个新的（最外层）定义。在本书的下一个部分，我们会利用这两种思想来创建新类型的程序。



第八部分

复合值的改变





封装

在设计交通信号灯的控制程序时，我们可能并不是只想控制一个交通信号灯，而是想控制多个交通信号灯。类似地，在设计管理电话号码的程序时，我们可能想要管理多本通讯录，而不止是一本。当然，我们可以复制交通信号灯控制器（或者通讯录管理器）的代码，并且为状态变量重命名，但是复制代码并不是好的选择。另外，我们可能想要控制很多个交通信号灯，而多次复制代码是不现实的。

正确的解决方法是使用抽象。这里，我们对包括通讯录管理和交通信号灯控制等在内的几个程序的实例进行抽象。因为涉及到了状态变量，这与本书第四部分中的抽象相比，虽然概念不同，但是思想，甚至技术都是相同的。我们用一个 local 表达式封装状态变量与函数，这给了我们创建任意多个程序版本的能力。本章第一节，学习如何封装状态变量，第二节，对此进行讨论。

39.1 状态变量的抽象

假设我们要把图 36.2 中的程序转化为一个管理多个（模拟的）交通信号灯的程序。一个模拟的管理者应当能够独立控制每一个交通信号灯。事实上，这个管理者还应当能够添加或者关闭交通信号灯，同时保持系统的其他部分不变。

按照经验，我们知道每个交通信号灯需要两个定义：

1. 状态变量 *current-color*，它记录信号灯当前的颜色；
2. 服务函数 *next*，它依照交通法规转换交通信号灯的状态。

对于图形模拟来说，服务函数还需要重新绘制交通信号灯，使用户可以观察当前的颜色。最后，每个交通信号灯在画布上都有一个特定的位置：

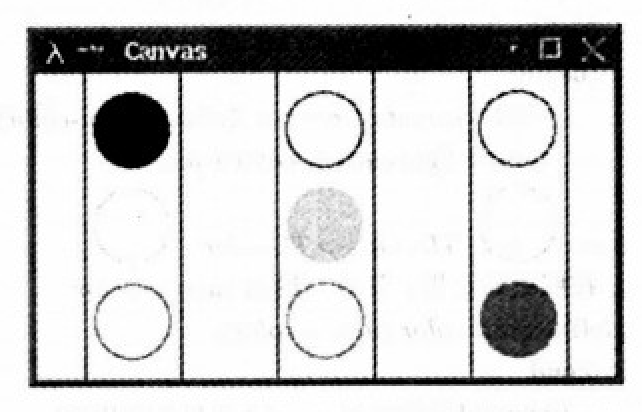


图 36.2 中的样本程序只能处理单一的交通信号灯，而且缺少绘图操作。现在的问题是修改它，使之能够根据需要处理多个交通信号灯，其中每一个交通信号灯都有着自己的状态变量与转换函数，还有着自己的位置。如果复制图 36.2 中的定义，再加上处理画布的定义，那么不同的（信号灯）实例只在一个方面不同：即交通信号灯位置。这告诉我们，应当开发一个抽象函数，它在不同的位置建立交通信号灯，并管理这些交通信号灯。

因为原来的程序由多个最外层的定义组成，所以我们使用 22.2 节中的诀窍。该诀窍指出，一个函数

应该用 local 表达式把定义包围起来。如果局部定义包含了状态变量，如上例，我们就说对定义进行了封装。封装强调了抽象函数对程序其他部分而言状态变量的隐藏。具体来说，就是通过把状态变量放入 local 表达式，保证了它只能按照服务管理进行改变，而不能被任意赋值修改。由此，定义封装和抽象同时进行，而程序员必须把这记在脑中。

下一步，我们要考虑这个函数应该做些什么，以什么为输入，返回什么，如果有效果的话，是什么样的。先考虑它的名字。我们把这个新的函数称为 *make-traffic-light*；毕竟，这个抽象程序的用途是建立一个模拟的交通信号灯。另外，按照抽象诀窍，抽象函数的输入是个实例特定的值。交通信号灯的特定值就是它在画布上的位置；为了明确，我们再加上它的实际地址。

每一次使用 *make-traffic-light* 都应该建立一个新的交通信号灯，并且提供切换这个信号灯状态的操作。前一部分表示了它的效果，具体来说，这个函数应当初始化状态变量，并在画布上的指定位置绘制出交通信号灯的初始状态；后一部分则描述了它的返回值：一个切换交通信号灯状态的函数。

```
;; 视图：
;; draw-light : TL-color number -> true
;; 在画布上（重新）绘制交通信号灯
(define (draw-light current-color x-posn) ...))

;; 模型：
;; make-traffic-light : symbol number -> (-> true)
;; 以(make-posn x-posn 0)为左上角，建立一个红色的信号灯
;; 效果：在画布上绘制交通信号灯
(define (make-traffic-light street x-posn)
  (local (;; current-color : TL-color
          ;; 记录交通信号灯当前的颜色
          (define current-color 'red)

            ;; init-traffic-light : -> true
            ;; （重新）把 current-color 设为'red，并（重新）建立视图
           (define (init-traffic-light)
             (begin
               (set! current-color 'red)
               (draw-light current-color x-posn)))

          ;; next : -> true
          ;; 效果：改变 current-color，'green 变为'yellow，
          ;; 'yellow 变为'red，'red 变为'green
          (define (next)
          (begin
               (set! current-color (next-color current-color))
               (draw-light current-color x-posn)))

         ;; next-color : TL-color -> TL-color
         ;; 按照交通法规，计算后继的 current-color
         (define (next-color current-color)
           (cond
             [(symbol=? 'green current-color) 'yellow]
             [(symbol=? 'yellow current-color) 'red]
             [(symbol=? 'red current-color) 'green])))
    (begin
      ;; 初始化，并返回 next
      (init-traffic-light)
      next)))
```

图 39.1　管理多个交通信号灯

图 39.1 给出了交通灯模拟器的框架，包括了 *make-traffic-light* 的完整定义。这个模拟器由模型与视图组成。视图被称作 *draw-light*，这里只给出它的框架；视图的完整定义作为练习题。

make-traffic-light 的定义是一个普通函数的定义，它使用一个 local 定义来设置状态变量、初始化函数和改变状态的函数。local 表达式的主体使用这个初始化函数，然后返回 *next*。

使用 *make-traffic-light*，我们可以建立多个独立的交通信号灯，或者建立交通信号灯的集合，也可以随时添加信号灯。首先建立一个充分大的画布：

```
;; 先建立画布
(start 300 160)
接着，按照需要，调用 make-traffic-light:
;; lights : (listof traffic-light)
;; 管理 Sunrise 沿线的信号灯
(define lights
  (list(make-traffic-light 'sunrise@rice 50)
       (make-traffic-light 'sunrise@cmu 150)))
```

这里，我们将 *lights* 定义为两个交通信号灯的表。每一个交通信号灯是一个函数，所以 *lights* 代表了两个函数的表。

在建立了交通信号灯之后，我们可以按意愿改变它们的状态。要做到这一点，必须记住每个交通信号灯都是由一个函数代表的，这种函数不读入参数，但返回 true。它的效果是改变隐含的状态变量，并在画布上画出。在我们的例子中，可以这样使用 Interactions 窗口：

```
> ((second lights))
true
> (andmap (lambda (a-light) (a-light)) lights)
true
```

第一个交互提取出 *lights* 的第二个元素，并调用它。这会把位于'sunrise@cmu 的信号灯设为绿色。第二个交互改变 *lights* 中所有元素的状态。

每一次调用 *make-traffic-light*，都会把 local 定义的变量改名，再提取为最外层的定义。因为前述的 define 包含了两次对 *make-traffic-light* 的调用，所以它会在计算中建立了两个 local 局部定义的函数副本和状态变量副本：

```
;; 'sunrise@rice 的定义
(define current-color@rice 'red)

(define (init-traffic-light@rice)
  (begin
    (set! current-color@rice 'red)
    (draw-light current-color@rice 50)))

(define (next@rice) ...)

(define (next-color@rice current-color) ...)
;; 'sunrise@cmu 的定义
(define current-color@cmu 'red)

(define (init-traffic-light@cmu)
  (begin
```

```
      (set! current-color@cmu 'red)
      (draw-light current-color@cmu 150)))
  (define (next@cmu) ...)
  (define (next-color@cmu current-color) ...)
  (define lights
    (list next@rice
          next@cmu))
```

init-traffic-light 新的最外层定义显示了重命名是如何确保它们中的一个处理'sunrise@rice 而另一个处理'sunrise@cmu。

习题

习题 39.1.1 前述第二个交互的效果是什么?

习题 39.1.2 在手工计算程序中，填写 *next@rice* 和 *next@cmu* 的主体，然后在这些定义的环境下计算((second *lights*))。

习题 39.1.3 开发函数 *draw-light*，它实现图 39.1 中交通信号灯模拟器的视图部分。每一个交通信号灯都应该与画布一样高，由左右两条实心线描绘。这表明一个信号灯的尺寸是:

```
(define WIDTH 50)
(define RADIUS 20)
(define DISTANCE-BETWEEN-BULBS 10)
;; 最小的画布高度
(define HEIGHT
  (+ DISTANCE-BETWEEN-BULBS
     (* 2 RADIUS)
     DISTANCE-BETWEEN-BULBS
     (* 2 RADIUS)
     DISTANCE-BETWEEN-BULBS
     (* 2 RADIUS)
     DISTANCE-BETWEEN-BULBS))
```

开发必需的、与交通信号灯程序其余部分分离的定义，然后用 local 创建一个单独的定义。

现在，假设我们要提供另一个服务，重新设置单一的交通信号灯。也就是说，除了转换当前的颜色之外，还需要一个操作，可以把某个交通信号灯设为红色。这样的函数已经存在了: *init-traffic-light*，它把 *current-color* 设为'red，并重新在画布上绘制图像。但是，我们无法获取 *init-traffic-light*，因为它是在 *make-traffic-light* 的 local 表达式中定义的。如果想要获得这个函数，它必须成为 *make-traffic-light* 的返回值，就像 *next* 一样。

要使 *next* 和 *init-traffic-light* 都成为 *make-traffic-light* 的返回值，需要一种方法，把这两个函数结合为一个单一的值。既然在 Scheme 中函数是值，我们可以用表、结构体，或者是向量来结合两个函数。另一种可能的方法是用第三个函数来结合这两个函数。这里我们就讨论这第三种可能，因为在管理状态变量和服务方面，它是一种重要的技术。

我们把这类新函数称为 *service-manager*，因为它隐藏并管理了实现服务的函数，该函数接受两个符号:

1.'next，它表明(*next*)应当被计算，

2.'reset，它表明(*reset*)应当被计算。

另外，这个函数是修改后的 *make-traffic-light* 的返回值。

图 39.2 给出了修改后的 *make-traffic-light* 的定义。因为一个错误的操作可能把函数作用于不适当的参数，所以 *service-manager* 是一个带检查的函数（参见第 7.5 节）。如果输入是'next 或'reset 以外的符号，它就产生一个错误消息。

```
;; make-traffic-light : symbol number -> (symbol -> true)
;; 以(make-posn x-posn 0)为左上角，建立一个红色的信号灯
;; 效果：　在画布上绘制交通信号灯
(define (make-traffic-light street x-posn)
  (local (;; 模型:
          ;; current-color : TL-color
          ;; 记录交通信号灯当前的颜色
          (define current-color 'red)

          ;; init-traffic-light : -> true
          ;; （重新）把 current-color 设为'red，并（重新）建立视图
          (define (init-traffic-light) ...)

          ;; next : -> true
          ;; 效果：　改变 current-color，'green 变为'yellow，
          ;; 'yellow 变为'red，'red 变为'green
          (define (next) ...)

          ;; next-color : TL-color -> TL-color
          ;; 按照交通法规，计算后继的 current-color
          (define (next-color current-color) ...)
          ;; service-manager : (symbol -> true)
          ;; 调用 next 或者 init-traffic-light
          (define (service-manager msg)
            (cond
              [(symbol=? msg 'next) (next)]
              [(symbol=? msg 'reset) (init-traffic-light)]
              [else (error 'traffic-light "message not understood")])))
    (begin
      ;; 初始化并返回 service-manager
      (init-traffic-light)
      service-manager)))
```

图 39.2　管理多个交通信号灯，提供重置服务

使用新的 *make-traffic-light* 函数就和使用原来的函数一样：

```
;; 先建立画布
(start 300 160)

;; lights : (listof traffic-light)
;; 管理 Sunrise 沿线的信号灯
(define lights
  (list (make-traffic-light 'sunrise@rice 50)
        (make-traffic-light 'sunrise@cmu 150)))
```

不过，在现在的返回值中，每个交通信号灯都用一个处理符号的函数表示：

```
> ((second lights) 'next)
```

```
true
> (andmap (lambda (a-light) (a-light 'reset)) lights)
true
```

第一个交互把标号为'sunrise@cmu 的信号灯从初始的红色转换为'green。第二个交互把所有信号灯的状态都改回'red。

习题

习题 39.1.4 使用习题 39.1.3 中的函数，完成图 39.2 中程序的定义。然后使用 DrScheme 的 Interactions 窗口对交通信号灯进行转换和重置。

习题 39.1.5 手工计算上述程序，确认标号为'sunrise@rice 的信号灯直接从'green 转换回'red，跳过了'yellow。

对于本书第七部分通讯录的例子来说，管理两项服务的需要就更明显了。通过例子激发的想法是，用户可以通过两种不同的服务来存取一个状态变量：*add-to-address-book* 用来添加新的条目，*lookup* 用来在电话本中查找某个给定的人名。按照封装诀窍，我们必须：

1. 定义一个函数 *make-address-book*，其主体是一个 local 表达式；
2. 把定义放入这个 local 表达式；
3. 引入一个名为 *service-manager* 的函数，用来管理这两项服务。

现在，我们已经在严格的控制下完成了前两个步骤；不过，第三步就较为复杂了，与前一个例子不同，这两个实现服务的函数读入的参数数目不同，而且返回不同类型的返回值。

我们先就 *service-manager* 的输入取得一致。为了方便记忆，添加电话号码所用的符号是'add，而查找给定人名号码所用的符号是'search。这表明了如下的模板：

```
(define (service-manager msg)
  (cond
    [(symbol=? msg 'add) ... A ...]
    [(symbol=? msg 'search) ... B ...]
    [else (error 'address-book "message not understood")]))
```

问题是，我们不清楚怎样用正确的 Scheme 表达式替换 *A* 和 *B* 才能计算出适当的返回值，并产生合适的效果。对于 *A*，不仅需要 *msg*，还需要一个人名和一个电话号码。对于 *B*，只需要一个人名。

一种解决方法是返回一个函数，让这个函数来读入额外的参数，并执行相应的计算。换一种说法，*service-manager* 现在是一个读入两个符号返回一个函数的函数。因为以前还没有遇到过这种类型的返回值，所以我们引入一种新的数据定义形式：

> address-book（通讯录）是一个界面：
>
> 1. 'add :: symbol number -> void
> 2. 'search :: symbol -> number

这个数据定义中提到了界面的概念，界面是一个函数，它读入有限多种符号，挨个返回不同类型的函数。因为这种类型的函数与以前所看到过的函数有着本质上的不同，所以使用了一个不同的名字。

现在可以写出合约与用途说明了：

```
;; service-manager : address-book
;; 管理通讯录的添加与查找
(define (service-manager msg) ...)
```

要定义这个函数需要区别两种情况。对于'add 的情况，要返回的东西显然就是 *add-to-address-book*。

对于'search 的情况，我们需要一个函数，它读入一个符号，然后把 *lookup* 作用于这个符号和 local 局部定义的 *address-book*。使用 lambda，我们可以快速建立这样一个函数：

```
(lambda (name)
  (lookup name address-book))
```

既然这个函数是一个值，它自然就是'search 的答案。

图 39.3 给出了 *make-address-book* 完全的定义。现在，这个定义是标准的了，它由一个 local 表达式组成，而这个 local 表达式又以 local 局部定义的 *service-manager* 作为返回值。这里并不需要初始化程序，因为惟一的状态变量直接被初始化，而且也没有图形显示。

```
;; make-address-book : string -> address-book
;; 创建一个函数，管理一本隐藏的通讯录所有的服务
(define (make-address-book title)
  (local ((define-struct entry (name number))
          ;; address-book : (listof (list name number))
          ;; 记录人名—电话号码关联的一个表
          (define address-book empty)

         ;; add-to-address-book : symbol number void
          ;; 效果： 添加一个人名—电话号码到相应的 address-book 中
         (define (add-to-address-book name phone)
              (set!  address-book  (cons  (make-entry  name  phone)
address-book)))

         ;; lookup : symbol (listof (list symbol number)) -> number or false
         ;; 在 address-book 中查找 name 所对应的电话号码
         (define (lookup name ab)
           (cond
             [(empty? ab) false]
             [else (cond
                     [(symbol=? (entry-name (first ab)) name)
                 (entry-number (first ab))]
                     [else (lookup name (rest ab))])]))

         ;; service-manager : address-book object
         ;; 管理通讯录的添加与查找
         (define (service-manager msg)
           (cond
             [(symbol=? msg 'add)
            add-to-address-book]
             [(symbol=? msg 'search)
            (lambda (name)
           (lookup name address-book))]
             [else (error 'address-book "message not understood")])))
    service-manager))
```

图 39.3　管理多本通讯录

要使用一本通讯录，首先用 *make-address-book* 建立它：

```
;; friends : an address book
;; 记录朋友的通讯录
(define friends
  (make-address-book "Friends of Charles"))

;; business : an address book
;; 记录生意伙伴的通讯录
(define business
  (make-address-book "Colleagues @ Rice, Inc."))
```

这两个定义建立了两本不同的通讯录，一本是朋友们的记录，另一本是生意伙伴的记录。

接着，我们向通讯录中添加人名和电话号码，或者查找电话号码：

```
> ((friends 'add) 'Bill 2)
> ((friends 'add) 'Sally 3)
> ((friends 'add) 'Dave 4)
> ((business 'add) 'Emil 5)
> ((business 'add) 'Faye 18)
```

这里，我们向名为 *friends* 的通讯录中添加三个条目，向名为 *business* 的通讯录中添加两个条目。

另外，*friends* 的工作分为两步。第一步是把 *friends* 作用于'add，给出（隐藏的）函数 *add-to-address-book*；第二步是把这个返回函数作用于一个人名与一个电话号码。按照类似的方式，查找一个电话号码也分两步。比方说，把 *friends* 作用于'search 给出一个读入人名的函数，接着将这个函数作用于一个符号：

```
> ((friends 'search) 'Bill)
2
> ((business 'search) 'Bill)
false
```

这两个调用说明，在 *friends* 中，'Bill 的电话号码是 2，而 *business* 中并没有'Bill 的号码。按照通讯录中的内容，这正是我们所期望的。当然，也可以在 Interactions 窗口中混合这两种操作，任意地添加和查找电话号码。

习题

习题 39.1.6 针对 *make-traffic-light* 修改版本的结果（参见图 39.2），开发一个界面定义。

习题 39.1.7 给出对 *friends* 和 *business* 的计算所建立的最外层定义。

在计算了五个'add 表达式之后，这些定义的状态是什么？在这个背景中计算((*friends* 'search) 'Bill)。

习题 39.1.8 设计 *gui-for-address-book*。这个函数读入一个字符串的表，对其中的每一个字符串建立一本通讯录。它还创建并显示一个通讯录的图形用户界面，该界面中有一个选择菜单，允许用户选择他们想要操作的通讯录。

39.2 封装练习

习题 39.2.1 开发程序 ***make-city***，该程序管理一组交通信号灯。它应当提供四项服务：

1. 添加一个带有标号（字符串）的交通信号灯；
2. 根据标号去除一个交通信号灯；
3. 转换某个给定标号的交通信号灯的状态；
4. 把某个给定标号的交通信号灯重设为红色。

提示： 前两项服务是直接提供的；后两项服务由模拟的交通信号灯实现。

完成上述程序之后，设计一个图形用户界面。

习题 39.2.2 设计 ***make-master***，该函数建立 37.1 节猜颜色游戏的一个实例，惟一的返回值是 ***master-check*** 函数。在游戏者猜出答案之后，这个函数应该简单地响应“game over”。如下是一段典型的对话：

```
> (define master1 (make-master))
> (master-check 'red 'red)
'NothingCorrect
> (master-check 'black 'pink)
'OneColorOccurs
...
```

将这段对话与 37.1 节中的一段对话相比较。

添加一项服务到 ***make-master***，它揭示出隐含的颜色。这样，一个对某个游戏感到厌倦的游戏者就可以找到答案了。

习题 39.2.3 开发 ***make-hangman***，该程序读入一个单词表，使用这个表建立一个刽子手游戏，并返回 ***hangman-guess*** 函数。游戏者可能会进行如下的对话：

```
> (define hangman-easy (make-hangman (list 'a 'an 'and 'able 'adler)))
> (define hangman-difficult (make-hangman (list 'ardvark ...)))
> (hangman-easy 'a)
"You won"
> (hangman-difficult 'a)
(list 'head (list '_ '_ '_ '_ '_ '_))
> ...
```

将这段对话与 37.2 节中的一段对话相比较。

添加一项服务到 ***make-master***，它揭示隐含的单词。

一个可选择的扩展是为这个程序装备一个图形用户界面和一张线条画（用图圈表示头部，用线条组合表示身体剩下的部位）。请尽可能重用现有的解决方案。

习题 39.2.4 设计 ***make-player***，该函数对 37.5 节中的函数进行抽象。使用该函数，我们可以建立多个在校园中闲逛的游戏者：

```
(define player1 (make-player 'BioEngineering))
(define player2 (make-player 'MuddBuilding))
...
```

make-player 的参数指定了游戏者的初始位置。

每一个（游戏者）实例都应该能够返回：

1．当前环境的图片；

2．可以到达的连通建筑物的表；

3．通过可用连接，到达另一地方的一个移动。

扩展：两个游戏者可能同时在同一幢建筑中，但是它们并不能进行交互。扩展这个游戏，使得在同一幢建筑中的两位游戏者可以以某种形式交互。

习题 39.2.4 开发程序 *moving-pictures*，它读入一个位置以及一张图片，即在第 6.6 节、第 7.4 节和第 10.3 节中定义的，图形的表（也可以参见第 21.4 节，关于移动图片的函数）。这个程序支持两种服务。第一种，它可以把图形放置在特定的位置。第二种，它可以把图片重新设置到初始给定的位置。



可改变的结构体



封装和管理状态变量类似于组成和管理结构体。在第一次调用包含对状态变量进行抽象的函数时，我们会提供某些变量的初始值。服务管理器提供对这些变量（当前）值的服务，这类似于在结构体中提取字段的值。那么，毫无奇怪，这种技术可以模拟 define-struct 定义的构造器和选择器。这种模拟自然使我们想到了引入可以修改结构体字段的值的函数。本章前两节说明了这种思想背后的细节；最后一节把它推广到向量上。

40.1 由函数得出结构体

观察图 40.1，它的左边只有一行，是 *posn* 结构体的定义，右边是一个函数的定义，提供了几乎相同的所有服务。具体来说，这个定义提供了一个构造器和两个选择器，构造器读入两个值，构造出一个复合值，而两个选择器提取出这个复合值的构成成分的值。

```
(define-struct posn (x y))      (define (f-make-posn x0 y0)
                                  (local ((define x y0)
                                          (define y y0)
                                          (define (service-manager msg)
                                            (cond
                                              [(symbol=? msg 'x) x]
                                              [(symbol=? msg 'y) y]
                                              [else (error 'posn "...")])))
                                    service-manager))

                                (define (f-posn-x p)
                                  (p 'x))

                                (define (f-posn-y p)
                                  (p 'y))
```

图 40.1 一个类似于 posn 的函数

要理解为什么 *f-make-posn* 是一个构造器，而为什么 *f-posn-x* 和 *f-posn-y* 是选择器，我们来讨论它们是怎样工作的，并确认预期的等式成立。这里，我们就做这两件事，因为这种定义太不寻常了。

f-make-posn 的定义封装了两个变量和一个函数。这两个变量代表了 *f-make-posn* 的参数，而这个函数是一个服务管理器：当它被给定输入'x，它就返回 *x* 的值，当它被给定输入'y，它就返回 y 的值。在前面的章节中，我们可以写出了类似于

```
(define a-posn (f-make-posn 3 4))
```

```
(+ (a-posn 'x) (a-posn 'y))
```

的代码，来定义并计算 *f-make-posn*。既然（从结构体中）提取值是一种频繁的操作，图 40.1 引入了函数 *f-posn-x* 和 *f-posn-y*，它们执行这种计算。

第 8 章介绍结构体时，我们曾说选择器和构造器可以用等式来描述。对于 *posn* 的定义，两个相关的等式是：

```
  (posn-x (make-posn V-1 V-2))
= V-1
```

和

```
  (posn-y (make-posn V-1 V-2))
= V-2
```

其中 *V-1* 和 *V-2* 是任意的值。

要确认 *f-posn-x* 和 *f-make-posn* 的关系与 posn-x 和 make-posn 的关系相同，我们先来验证它们满足第一个等式：

```
  (f-posn-x (f-make-posn 3 4))
= (f-posn-x (local ((define x 3)
                    (define y 4)
                    (define (service-manager msg)
                      (cond
                 [(symbol=? msg 'x) x]
                 [(symbol=? msg 'y) y]
                 [else (error 'posn "...")])))
      service-manager))
= (f-posn-x service-manager)
  ;; 添加到最外层定义：
  (define x 3)
  (define y 4)
  (define (service-manager msg)
    (cond
      [(symbol=? msg 'x) x]
      [(symbol=? msg 'y) y]
      [else (error 'posn "...")]))
= (service-manager 'x)
= (cond
    [(symbol=? 'x 'x) x]
    [(symbol=? 'x 'y) y]
    [else (error 'posn "...")])
= x
=3
```

证明 *f-posn-y* 和 *f-make-posn* 满足类似的等式留作练习。

习题

习题 40.1.1　结构体的模拟没有提供哪种功能？为什么没有提供？

习题 40.1.2　下面是 *posn* 结构体的另一种实现：

```
(define (ff-make-posn x y)
  (lambda (select)
```

```
    (select x y)))
(define (ff-posn-x a-ff-posn)
  (a-ff-posn (lambda (x y) x)))
(define (ff-posn-y a-ff-posn)
  (a-ff-posn (lambda (x y) y)))
```

在这个环境下，计算(ff-posn-x (ff-make-posn V-1 V-2))。这个计算示范了什么？

习题 40.1.3　说明怎样用函数实现下列结构体的定义：

1. **(define-struct** *movie* (*title producer*))
2. **(define-struct** *boyfriend* (*name hair eyes phone*))
3. **(define-struct** *cheerleader* (*name number*))
4. **(define-struct** *CD* (*artist title price*))
5. **(define-struct** *sweater* (*material size producer*))

从中选择一个，示范预期的法则是成立的。

40.2　可变的函数结构体

第 39 章以及 40.1 节表明结构体是可变的。更确切地说，可以改变结构体中某个字段的值。第 39 章介绍了服务管理器，它隐藏了状态变量，而不仅仅是普通的变量定义。图 40.2 显示，图 40.1 中定义的一个微小改变是如何把 local 局部定义的变量转化为状态变量的。这个修改后的服务管理器为每个状态变量都提供两种操作：一个用来读出当前值，另一个用来改变当前值。

```
(define (fm-make-posn x0 y0)
  (local ((define x y0)
          (define y y0)
          (define (service-manager msg)
            (cond
              [(symbol=? msg 'x) x]
              [(symbol=? msg 'y) y]
              [(symbol=? msg 'set-x) (lambda (x-new) (set! x x-new))]
              [(symbol=? msg 'set-y) (lambda (y-new) (set! y y-new))]
              [else (error 'posn "...")])))
    service-manager))

(define (fm-posn-x p)
  (p 'x))

(define (fm-posn-y p)
  (p 'y))

(define (fm-set-posn-x! p new-value)
  ((p 'set-x) new-value))

(define (fm-set-posn-y! p new-value)
  ((p 'set-y) new-value))
```

图 40.2　一个带变化器的 posns 实现

考虑如下的定义和表达式：

```
(define a-posn (fm-make-posn 3 4))

(begin
  (fm-set-posn-x! a-posn 5)
  (+ (posn-x a-posn) 8))
```

对它们进行手工计算显示了结构体是怎样改变的，下面是（手工计算的）第一步：

```
  ...
=
  (define x-for-a-posn 3)
  (define y-for-a-posn 4)
  (define (service-manager-for-a-posn msg)
    (cond
      [(symbol=? msg 'x) x-for-a-posn]
      [(symbol=? msg 'y) y-for-a-posn]
      [(symbol=? msg 'set-x)
       (lambda (x-new) (set! x-for-a-posn x-new))]
      [(symbol=? msg 'set-y)
       (lambda (y-new) (set! y-for-a-posn y-new))]
      [else (error 'posn "...")]))
  (define a-posn service-manager-for-a-posn)
  (begin
    (fm-set-posn-x! a-posn 5)
    (+ (posn-x a-posn) 8))
```

上述代码重新命名了局部定义，并将它提取出 *fm-make-posn* 的定义。因为函数定义在剩下的计算中并不改变，所以我们只集中于变量的定义：

```
  (define x-for-a-posn 3)
  (define y-for-a-posn 4)
  (begin
    (fm-set-posn-x! a-posn 5)
    (+ (posn-x a-posn) 8))

= (define x-for-a-posn 3)
  (define y-for-a-posn 4)
  (begin
    (fm-set-posn-x! service-manager-for-a-posn 5)
    (+ (posn-x a-posn) 8))

= (define x-for-a-posn 3)
  (define y-for-a-posn 4)
  (begin
    ((service-manager-for-a-posn 'set-x) 5)
    (+ (posn-x a-posn) 8))

= (define x-for-a-posn 3)
```

```
  (define y-for-a-posn 4)
  (begin
    (set! x-for-a-posn 5)
    (+ (posn-x a-posn) 8))

= (define x-for-a-posn 5)
  (define y-for-a-posn 4)
  (+ (posn-x a-posn) 8)
```

这里，*x-for-a-posn* 的定义已经按照预期的方式被修改过了。从此以后，所有对这个状态变量的引用，也就是（模拟的）*x* 字段 *a-posn*，都代表 5。即今后所有对 *x-for-a-posn* 的引用都返回 5。

习题

习题 40.2.1　开发如下结构体定义的一个函数表示：

```
(define-struct boyfriend (name hair eyes phone))
```

使得模拟的结构体的字段可以被改变。

习题 40.2.2　下面是习题 40.1.2 中基于函数的 posn 结构体修改后的实现：

```
(define (ffm-make-posn x0 y0)
  (local ((define x x0)
          (define (set-x new-x) (set! x new-x))
          (define y y0)
          (define (set-y new-y) (set! y new-y)))
  (lambda (select)
    (select x y set-x set-y))))

(define (ffm-posn-x a-ffm-posn)
  (a-ffm-posn (lambda (x y sx sy) x)))

(define (ffm-posn-y a-ffm-posn)
  (a-ffm-posn (lambda (x y sx sy) y)))

(define (ffm-set-posn-x! a-ffm-posn new-value)
  (a-ffm-posn (lambda (x y sx sy) (sx new-value))))

(define (ffm-set-posn-y! a-ffm-posn new-value)
  (a-ffm-posn (lambda (x y sx sy) (sy new-value))))
```

示范如何修改结构体(ffm-make-posn 3 4)，使 y 字段包含 5。

40.3　可变的结构体

Scheme 的结构体是可变的。事实上，在 Advanced Student Scheme 中，如

```
(define-struct posn (x y))
```

这样的结构体定义引入了六个而不只是四个基本操作

1. make-posn，构造器；
2. posn-x 和 posn-y，选择器；
3. posn?，判断器；
4. set-posn-x!和 set-posn-y!，变化器。

变化器是改变结构体内容的操作。

回忆一下，我们把结构体当作有隔间的方框。例如，结构体

```
(make-posn 3 4)
```

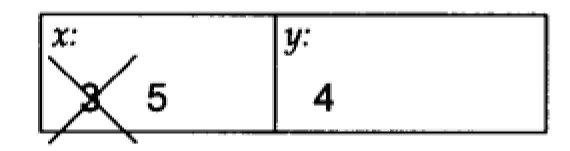
应当被视作有两个隔间的方框：

x: 3	y: 4

构造器建立隔间；选择器从特定的隔间中提取出值；判断器识别隔间；而变化器改变隔间的内容。换句话说，变化器对它的参数是有效果的；它的返回值是不可见的。用图像来描述，可以把

```
(define p (make-posn 3 4))
(set-posn-x! p 5)
```

这样一个表达式的计算想象为一个方框，把其中旧的 *x* 值删除，并插入一个新的 *x* 值到同一个方框中：



考虑以下定义：

```
(define-struct star (name instrument))
(define p (make-star 'PhilCollins 'drums))
```

考虑下列表达式的计算与效果：

```
(begin
  (set-star-instrument! p 'vocals)
  (list (star-instrument p)))
```

按照解释，第一个子表达式修改名为 *p* 的 *star* 结构体的 *instrument* 字段；第二个子表达式返回一个表，表中有一个元素，是名为 *p* 的结构体的 *instrument* 字段的当前值。类似于 40.2 节，计算过程如下：

```
  (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (begin
    (set-star-instrument! p 'vocals)
    (list (star-instrument p)))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (begin
    (void)
    (list (star-instrument p)))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
```

```
(list 'vocals)
```

其中第一个步骤改变 p 的定义中一个部分的值，第二个步骤提取出 *instrument* 字段的当前值，并把它放入一个表中。

在结构体中引入变化器需要计算规则系统作两点改变：

1. 每一个构造器表达式用一个对最外层是新的、惟一的名字添加一个定义，除非它已经在一个定义中出现过。[1]

2. 代表结构体的名字是值。

如果把所有的结构体都看作管理服务的函数，例如访问某个字段的当前值、修改字段等，就可以理解这些改变。毕竟，local 函数定义也用独特的名字创建最外层定义，而且函数的名字也是值。

使用这两条新的规则可以更深入地研究变化器不寻常的行为。下面是第一个例子：

```
(define-struct star (name instrument))

(define p (make-star 'PhilCollins 'drums))

(define q p)

(begin
  (set-star-instrument! p 'vocals)
  (list (star-instrument q)))
```

它与前一个定义有两方面的不同。第一点，它定义 q 为 p，第二点，begin 表达式的第二个子表达式引用 q 而不是 p。检查一下我们对计算过程的理解：

```
  (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q p)
  (begin
    (set-star-instrument! p 'vocals)
    (list (star-instrument q)))

 =(define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q p)
  (begin
    (void)
    (list (star-instrument q)))
```

照旧，第一步是改变 p 的定义的一个部分，第二步是访问 q 的当前值：

```
   ...
 = (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q p)
  (list (star-instrument p))
```

[1] 为简单起见，我们就使用这一简单的近似。当程序也使用 set!表达式时，我们必须依赖于 8.3 节的详细介绍来完整地理解它的行为。也请参见 40.5 节中关于这一主题的介绍。

```
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q p)
  (list 'vocals)
```

因为 *q* 是 *p*，而 *p* 的 *instrument* 字段的当前值是'vocals，所以返回值还是(list 'vocals)。

我们刚才所看到的就是共享的效果（变化器的效果），这意味着对结构体的修改不止一处地影响了程序。正如第二个例子所示的，共享在表中也是可见的：

```
(define-struct star (name instrument))


(define q (list (make-star 'PhilCollins 'drums)))


(begin
  (set-star-instrument! (first q) 'vocals)
  (list (star-instrument (first q))))
```

这里，*q* 的定义的右部是一个表达式，它惟一的子表达式不是一个值。更具体地说，它是一个结构体表达式，必须这样计算：

```
  ...
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q (list p))
  (begin
    (set-star-instrument! (first q) 'vocals)
    (list (star-instrument (first q))))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q (list p))
  (begin
    (set-star-instrument! p 'vocals)
    (list (star-instrument (first q))))
```

因此第一步是要引入一个新的定义，并且选用 *p* 作它的名字；第二步把(first *q*)替换为 *p*，因为 *q* 是一个表，其中只包含一个元素：*p*，余下的计算几乎与前面一样：

```
  ...
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q (list p))
  (begin
    (void)
    (list (star-instrument (first q))))
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q (list p))
  (list (star-instrument p))

= (define-struct star (name instrument))
```

```
(define p (make-star 'PhilCollins 'vocals))
(define q (list p))
(list 'vocals)
```

最后，效果可以被不同的表中的元素共享。观察这个程序的第三个变体：

```
(define-struct star (name instrument))

(define q (list (make-star 'PhilCollins 'drums)))

(define r (list (first q) (star-instrument (first q))))

(begin
  (set-star-instrument! (first q) 'vocals)
  (list (star-instrument (first r))))
```

新的定义引入变量 r，它代表了一个表，表中包含两个元素。使用新的规则来确定这个程序的返回值和效果：

```
...
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q (list p))
  (define r (list (first q) (star-instrument (first q))))
  (begin
    (set-star-instrument! (first q) 'vocals)
    (list (star-instrument (first r))))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q (list p))
  (define r (list p (star-instrument p)))
  (begin
    (set-star-instrument! (first q) 'vocals)
    (list (star-instrument (first r))))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'drums))
  (define q (list p))
  (define r (list p 'drums))
  (begin
    (set-star-instrument! (first q) 'vocals)
    (list (star-instrument (first r))))
```

照旧，第一个步骤为新的 *star* 结构体引入一个定义，第二和第三个步骤建立名为 r 的表，表中包含新建立的结构体 p 以及它的 *instrument* 的当前值**'vocals**。

下一个步骤是选出 q 的第一个元素，并修改它的 *instrument* 字段：

```
...
= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
```

```
  (define q (list p))
  (define r (list p 'drums))
  (begin
    (void)
    (list (star-instrument (first r))))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q (list p))
  (define r (list p 'drums))
  (list (star-instrument p))

= (define-struct star (name instrument))
  (define p (make-star 'PhilCollins 'vocals))
  (define q (list p))
  (define r (list p 'drums))
  (list 'vocals)
```

因为 *r* 的第一个元素包含了 *p*，并且 *p* 的 *instrument* 字段还是'vocals，所以这里的返回值还是(list 'vocals)。但是，这个程序还是包含了一些有关'drums，即 *star* 结构体原来值的信息。

总而言之，变化器向我们提供了比构造器和选择器更强的能力。除了仅仅建立新的结构体，并使用它们的内容之外，我们现在可以改变结构体的内容，同时保持结构体不变。接下来，我们必须要思考这对于程序设计来说意味着什么。

习题

习题 40.3.1 给出下列结构体定义所引入的变化器的名字：

1. **(define-struct** *movie* (*title producer*))
2. **(define-struct** *boyfriend* (*name hair eyes phone*))
3. **(define-struct** *cheerleader* (*name number*))
4. **(define-struct** *CD* (*artist title price*))
5. **(define-struct** *sweater* (*material size producer*))

习题 40.3.2 开发函数 *swap-posn*，该函数读入一个 *posn* 结构体，交换它的两个字段的值，它的返回值是**(void)**。

习题 40.3.3 开发函数 *one-more-date*，该函数读入一个 *girlfriends* 结构体，把 *number-past-dates* 字段的内容增加 1。这个结构体的定义是：

```
(define-struct girlfriends (name hair eyes number-past-dates))
```

函数的返回值是**(void)**。

习题 40.3.4 一步一步地计算下列表达式：

```
(define-struct cheerleader (name number))

(define A (make-cheerleader 'JoAnn 2))

(define B (make-cheerleader 'Belle 1))

(define C (make-cheerleader 'Krissy 1))
```

```
(define all (list A B C))

(list
  (cheerleader-number (second all))
  (begin
    (set-cheerleader-number! (second all) 17)
    (cheerleader-number (second all))))
```

在最后的程序中，用下划线标出与最初的程序不同的定义。

习题 40.3.5　计算如下程序：

```
(define-struct CD (artist title price))

(define in-stock
  (list
    ((make-CD 'R.E.M "New Adventures in Hi-fi" 0)
     (make-CD 'France "simple je" 0)
     (make-CD 'Cranberries "no need to argue" 0))))

(begin
  (set-CD-price! (first in-stock) 12)
  (set-CD-price!(second in-stock) 19)
  (set-CD-price! (third in-stock) 11)
  (+(CD-price (first in-stock))
    (CD-price (second in-stock))
    (CD-price (third in-stock))))
```

给出每一个步骤。

40.4　可变的向量

回忆第 29 章，其中说到向量类似于结构体，也是复合值。要从结构体中提取出值，程序使用选择器操作。要从向量中提取出值，程序使用自然数作为下标。因此，处理向量的函数需要使用辅助函数，对向量和自然数进行处理。

毫不奇怪，向量与结构体类似，也是可变的复合值。向量惟一的变化器是 vector-set!，这个函数读入一个向量、一个下标和一个值。因此，如下的程序计算出'blank：

```
(define X (vector 'a 'b 'c 'd))

(begin
  (vector-set! X 0 'blank)
  (vector-set! X 1 'blank)
  (vector-set! X 2 'blank)
  (vector-set! X 3 'blank)
  (vector-ref X 2))
```

上述四个 **vector-set!**表达式改变了 *X* 的值，使它的所有四个字段都包含'blank。最后一个表达式提取

出其中一个字段的值。

一般来说，关于可变向量的计算就类似于可变结构体的计算。特别是，vector 表达式引入一个新的定义：

```
  (list (vector 1 2 3))
= (list v)
  ;; 添加为最外层定义：
  (define v (vector 1 2 3))
```

变量名 *v* 是新的，而且是惟一的。类似地，vector-set!表达式修改向量定义的一个部分。

```
  (set-vector! (vector 1 2 3) 0 'a)

= (define v (vector 1 2 3))
  (set-vector! v 0 'a)

= (define v (vector 'a 2 3))
  (void)
```

最后，向量的效果也是共享的，就像结构体的效果一样。

习题

习题 40.4.1 计算如下程序：

```
(define X (vector 0 0 0 0))

(define Y X)

(begin
  (vector-set! X 0 2)
  (vector-set! Y 1 (+ (vector-ref Y 0) (vector-ref Y 1)))
  (vector-ref Y 1))
```

给出所有的步骤。

习题 40.4.2 开发函数 *clear*，该函数读入一个包含三个位置的向量，把它们都设为 0。

习题 40.4.3 开发函数 *swap*，该函数读入一个包含两个位置的向量，交换这两个位置的值。

习题 40.4.4 用函数

```
;; board-flip! : board N N -> boolean
;; 对 a-board 中下标为 i, j 的棋盘位置求反
(define (board-flip! a-board i j) ...)
```

扩展习题 29.3.14 中的棋盘表示。

不要忘了设计例子，并对函数进行测试。

40.5 改变变量与改变结构体

结构体的变化器和 set!表达式之间是有关系的。事实上， 40.2 节就是用后者来解释前者的效果的。

不过，这其中还有着重大的差别，程序设计者必须要理解这个差别。我们从语法开始：

```
(set! <variable> <expression>)          set-<structure-tag>-<field>!
```

set!表达式由两个部分组成：一个变量和一个表达式。变量是固定的，它永远不会被计算；表达式会被计算。与此相反，结构体的变化器是一个函数。就这一点而言，它是一个值，程序可以调用这个值（把它作用于两个参数），可以把它传递给其他的函数，可以把它存放入结构体。结构体的变化器是响应结构体的定义而创建的，就像结构体的构造器和选择器一样。

接下来，我们必须考虑辖域问题（参见第 18.2 节）。set!表达式包含了一个变量。要使 set!表达式有效，这个变量必须被绑定。在 set!表达式的变量和它的绑定出现之间的连接是静态的，并且永远不能改变。

变化器的辖域就是它相应的 define-struct 的辖域。因此，在如下程序中：

```
(define-struct aaa (xx yy))
(define UNIQUE
  (local ((define-struct aaa (xx yy)))
    (make-aaa 'my 'world)))
...
```

带下划线的 define-struct 出现的辖域是有限的，而且它的辖域完全在最外层 define-struct 的辖域之内。这种辖域的一个结果就是，最外层的 define-struct 的变化器不能改变名为 *UNIQUE* 的结构体。这两个变化器是不相关的函数，它们只是碰巧具备了相同的名字；计算 local 表达式的规则规定我们必须为其中的一个改名。

为了突出语法和辖域之间的区别，观察下面两个表面上类似的程序：

```
(define the-point (make-posn 3 4))      (define the-point (make-posn 3 4))
(set! x 17)                             (set-posn-x! the-point 17)
```

左边的程序是非法的，因为 set!表达式中的 *x* 是一个未绑定的变量。右边的程序是完全合法的：它引用了 *posn* 结构体中的 *x* 字段。

set!表达式与变化器之间最大的差别在于它们的语义。我们通过研究两个例子来理解它们之间根本的区别。第一个例子说明，看上去相似的表达式是怎样以根本上不同的方式计算的：

```
(define the-point (make-posn 3 4))      (define the-point (make-posn 3 4))
(set! the-point 17)                     (set-posn-x! the-point 17)
```

左边的程序由一个 *the-point* 定义和一个对 *the-point* 的赋值组成；右边的程序由同一个 *the-point* 的定义开始，后跟一个对变化器的调用。对这两个程序的计算都会影响变量的定义，但是方式不同：

```
(define the-point 17)                   (define the-point (make-posn 17 4))
(void)                                  (void)
```

在左边，*the-point* 现在代表一个数；在右边，它还是一个 *posn* 结构体，但是 *x* 字段中有了一个新的值。更一般地说，set!表达式改变一个定义右部的值，而对变化器的调用只改变出现在一个定义的右部的结构体中一个字段的值。

第二个例子说明对变化器的调用是怎样计算参数的，而这与 set!表达式的情况不同：

```
(define the-point (make-posn 3 4))      (define the-point (make-posn 3 4))
(define an-other (make-posn 12 5))      (define an-other (make-posn 12 5))

(set! (cond                             (set-posn-x!
    [(zero? (point-x the-point))          (cond
          the-point]                         [(zero? (point-x the-point))
```

```
        [else
          an-other])
        1)
```

```
                                the-point]
                               [else
                                an-other])
                              1)
```

尽管左边的程序是非法的，因为 set!表达式必须在第二个位置上包含一个固定的变量，但是右边的程序是合法的。对右边的程序计算会把 *an-other* 的 *x* 字段改为 1。

最后，*mutator*（变化器）是值，这意味着函数可以读入一个变化器，然后调用它：

```
;; set-to-2 : S-mutator S-structure -> void
;; 通过 mutator，把 s 中的一个字段改为 2
(define (set-to-2 mutator s)
  (mutator s 2))

(define-struct bbb (zz ww))

(local ((define s (make-posn 3 4))
        (define t (make-bbb 'a 'b)))
  (begin
    (set-to-2 set-posn-x! s)
    (set-to-2 set-bbb-ww! t)))
```

函数 *set-to-2* 读入一个变化器和一个该变化可以修改的结构体。这个程序使用这个函数来修改一个 *posn* 结构体中的 *x* 字段以及一个 *bbb* 结构体中的 *ww* 字段。与此不同，如果我们把一个函数作用于 set!表达式，这个函数只会收到(void)。

set!和结构体变化器的混合使用：当一个程序既使用 set!表达式，又使用结构体的变化器的时候，计算规则就不能处理某些情况。具体来说，它们不能正确地解释共享。考虑如下程序片断：

```
(define the-point (make-posn 3 4))

(define another-point the-point)

(begin
  (set! the-point 17)
  (= (posn-x another-point) 3))
```

依照规则，这两个定义引用的是同一个结构体，其中第二个定义是间接引用。set!表达式改变了 *the-point* 所代表的东西，但是它不应该影响第二个定义。特别地，这个程序应当返回 true。如果简单地使用规则，就不能得出这个正确的结论。

对结构体准确的解释必须为每一个结构体构造器的调用引入一个定义，包括原来程序中那些定义的右部。我们会把新的定义放到定义序列的前部。另外，在新的定义中的变量必须是惟一的，这样它就不可能出现在某个 set!表达式中。我们会使用像 *struct-1*、*struct-2* 等等这样的变量，并且只让它们起这个作用。这些名字是值，也只有这些名字是值。

使用这个小小的改变，我们现在可以正确地计算这个程序片断：

```
(define the-point (make-posn 3 4))

(define another-point the-point)

(begin
```

```
    (set! the-point 17)
    (= (posn-x another-point) 3))

= (define struct-1 (make-posn 3 4))
  (define the-point struct-1)
  ;; 从这里开始计算:
  (define another-point the-point)
  (begin
    (set! the-point 17)
    (= (posn-x another-point) 3))

= (define struct-1 (make-posn 3 4))
  (define the-point struct-1)
  (define another-point struct-1)
  ;; 从这里开始计算:
  (begin
    (set! the-point 17)
    (= (posn-x another-point) 3))
```

这里，结构体被定义，原来的两个变量都引用这个新的结构体。余下的计算改变 *the-point* 的定义，但没有改变 *another-point* 的定义：

```
  ...
= (define struct-1 (make-posn 3 4))
  (define the-point 17)
  (define another-point struct-1)
  ;; 从这里开始计算:
  (begin
    (void)
    (= (posn-x another-point) 3))

= (define struct-1 (make-posn 3 4))
  (define the-point 17)
  (define another-point struct-1)
  ;; 从这里开始计算:
  (= (posn-x another-point) 3)
= (define struct-1 (make-posn 3 4))
  (define the-point 17)
  (define another-point struct-1)
  ;; 从这里开始计算:
  (= 3 3)
```

正如我们所预期的，最后的结果是 true。

修改后的计算规则比起原来的稍微麻烦一些，但是它们彻底地解释了 set!表达式与结构体变化器之间的差别，而这对于现代编程语言来说是一个基本的概念。



第41章 设计改变结构体的函数

前面两章介绍了可变结构体的思想，其中第 39 章学习了通过函数改变局部定义的变量的方法，第 40 章讨论了如何修改结构体。

本章学习这种新功能的使用。第 1 节讨论为什么程序要修改结构体，第 2 节回顾了现有的设计诀窍，研究如何应用于变化器，第 3 节讨论一些困难的事例，最后一节讨论 set!和结构体变化器之间的差别。

41.1 为什么改变结构体

只要调用结构体的构造器，就会创建一个新的结构体。有时候，这就是我们真正想要的。考虑这样一个函数，它读入一个职员记录表，返回电话簿条目表。职员记录可能会包含某个人的地址，包括电话号码、生日、婚姻状况、亲戚以及工资等。电话簿条目应当只包含人名和电话号码。这种类型的程序应当由输入表中每一个结构体生成一个确定的新结构体。

不过，在其他的场合，建立一个新的结构体并不符合我们的直觉。假设我们要给某人加工资。此刻，惟一可以完成这件事的方法是建立一个新的职员记录，其中包含所有原来的信息以及新的工资信息。或者，假设我们要更新我们的 PDA 中的电话簿。跟修改某个职员的工资等级的程序一样，这个更新电话簿的程序要建立一个新的电话簿条目。不过，事实上我们并不会建立一个新的职员记录，或者在电话簿中建立一个新的条目，而是会修改现有的职员记录以及电话簿中现有的条目。程序应该可以执行这种类型的修改任务，有了变化器，我们确实可以开发出这样的程序。

粗略地说，这些例子描述了两种情况。第一种情况，如果某个结构体对应于物理对象，而计算对应于修正行为，程序就可能需要结构体的变化器；第二种情况，如果结构体并不对应于某个物理对象，或者计算是从现有的信息中建立一种新类型的值，程序就应该建立一个新的结构体。这两条规则并不是泾渭分明的。我们常常会遇到这样的情况，此时这两种解决方案都是可行的。在这种情况下，我们必须考虑哪种方案更易于编程。如果其中的某一种解决方案更易于设计，这通常是创建一个新的结构体，我们就选择它。如果这种决定导致了性能瓶颈，也仅仅在此情况下，我们才选择另一种方案。

41.2 结构体的设计诀窍与变化器之一

令人惊讶地，用变化器来编写程序并不需要任何新的设计诀窍，只要被改变的字段总是包含原子值，原有的诀窍就可以完美地工作。设计不含变化器的程序需要把值结合起来，而设计含变化器的程序需要把效果结合起来。因此，关键是要在程序的合约中加上定义明确的效果说明，并构造能说明效果的例子。我们在第 36 章中已经练习过针对 set!表达式的这些行为了。在这一章，我们来学习怎样使设计诀窍与效果说明适应于结构体的变化器。要做到这一点，我们先来考虑一些例子，其中每一个例子都说明，某个适用的设计诀窍是如何帮助修改结构体或修改向量的函数的设计。

第一个例子是关于修改普通结构体的。假设给定了职工记录的结构体和数据定义：

```
(define-struct personnel (name address salary))
;; 职工记录（PR）是结构体：
;; (make-personnel n a s)
;; 其中 n 是符号，a 是字符串，s 是数。
```

读入这样一条记录的函数基于如下模板：

```
(define (fun-for-personnel pr)
  ... (personnel-name pr) ...
  ... (personnel-address pr) ...
  ... (personnel-salary pr) ...)
```

考虑一个增加工资字段的函数：

```
;; increase-salary : PR number -> void
;; 效果：修改 a-pr 的工资字段，加上 a-raise
(define (increase-salary a-pr a-raise) ...)
```

这个合约说明该函数读入一个 *PR* 和一个数。用途说明是一个效果说明，它解释了 *increase-salary* 的参数是怎样被修改的。

开发 *increase-salary* 的例子需要使用第 36 章中的技术。具体来说，必须要能够比较某个 *PR* 结构体（在函数调用）之前与之后的状态。

```
(local ((define pr1 (make-personnel 'Bob 'Pittsburgh 70000)))
  (begin
    (increase-salary pr1 10000)
    (= (personnel-salary pr1) 80000)))
```

当且仅当 *increase-salary* 对这个例子能正确地工作，这个表达式的返回值为 true。

现在可以使用模板和例子来定义这个函数：

```
;; increase-salary : PR number -> void
;; 效果：修改 a-pr 的工资字段，加上 a-raise
(define (increase-salary a-pr a-raise)
  (set-personnel-salary! a-pr (+ (personnel-salary a-pr) a-raise)))
```

跟往常一样，完整的定义使用了一些模板中的子表达式，但是模板提醒我们可以使用的信息是参数以及它们的组成部分；模板还提醒了我们可以修改的部分是使用选择器的字段。

习题

习题 41.2.1　构造一些 *increase-salary* 的例子，并测试这个函数。用布尔值表达式来表示这些测试。

习题 41.2.2　修改 *increase-salary*，使得它只接受在工资的 3%和 7%之间的 *a-raise* 值。否则，它就调用 error。

习题 41.2.3　开发 *increase-percentage*。该函数读入一个 *PR* 和一个在 3%和 7%之间的百分比。它增加 *PR* 中的工资字段，增加的量是百分比增量与 7000 中较少的那个。

习题 41.2.4　开发函数 *new-date*（新的约会对象）。它读入一条 *cheerleader*（啦啦队队长）记录，把一个约会对象添加到表的前部。下面是相关的定义：

```
(define-struct cheerleader (name dates))
;; cheerleader 是结构体：
```

```
;; (make-cheerleader n d)
;; 其中 n 是符号，d 是符号表。
```

例如，**(*make-cheerleader* 'JoAnn '(Carl Bob Dude Adam Emil))**是一条有效的 *cheerleader* 记录。开发一个例子，说明把**'Frank** 添加为约会对象意味着什么。

习题 41.2.5 回忆 *square*（正方形）的结构体定义：

```
(define-struct square (nw length))
```

相应的数据定义描述了 *nw* 字段总是 *posn* 结构体而 *length* 是数：

> square 是结构体：
> **(make-square** *p s***)**
> 其中 *p* 是 *posn* 而 s 是数。

开发函数 *move-square!*，该函数读入正方形 *sq* 和数 *delta*，修改 *sq*，把 *delta* 加到它的 x 坐标之上。查找出圆周的结构体和数据定义，并开发函数 *move-circle*，该函数类似于 *move-square*。

第二个例子使我们重新想起了处理共用体的函数的设计诀窍。我们遇到的第一个这种类型的例子是有关几何形状的。下面就是相关的数据定义：

> *shape* 是下列两者之一：
> 1. *circle*，
> 2. *square*。

关于 *circle* 和 *square* 的定义，请参见习题 41.2.5 或者本书的第一部分。

按照诀窍，处理 *shape* 的函数是由一个 cond 表达式组成的，其中包含两个子句：

```
(define (fun-for-shape a-shape)
  (cond
    [(circle? a-shape) ... (fun-for-circle a-shape) ...]
    [(square? a-shape) ... (fun-for-square a-shape) ...]))
```

每一个 cond 子句引用一个函数，其目的是处理相应的形状。

所以，假设要在 x 方向上把某个 *shape* 移动固定数量的像素。在本书的第一部分中，我们为这个目的新建一个结构体。现在，作为代替，我们可以使用 *circle* 和 *square* 结构体的变化器——也就是说，这个函数可以有一个效果：

```
;; move-shape! : shape number -> void
;; 效果：在 x 方向上把 a-shape 移动 delta 像素
(define (move-shape! a-shape)
  (cond
    [(circle? a-shape) (move-circle a-shape delta)]
    [(square? a-shape) (move-square a-shape delta)]))
```

函数 *move-circle* 和 *move-square* 是习题 41.2.5 的主题，因为它们读入并影响普通的结构体。

习题 41.2.6 构造 *move-shape!*的例子，并测试这个函数。用布尔值表达式表示测试！

习题 41.2.7 下列结构体定义描述某个音像商店所出售的商品：

```
(define-struct CD (price title artist))
(define-struct record (price antique title artist))
(define-struct DVD (price title artist to-appear))
(define-struct tape (price title artist))
```

给出 *music items*（音像商品）类型的数据定义，该定义由 *cd*、*record*、*dvd* 和 *tape* 组成。在每个对象中，价格（*price*）都必须是数。

开发程序 *inflate!*，该程序读入一个 *music-item* 和一个百分比，它的效果是按照这个百分比提高给定结构体的价格。

习题 41.2.8 开发一个程序，记录动物园中动物的给食情况。动物园中有三种类型的动物：大象、猴子和蜘蛛。每一只动物都有名字，每天都有两个给食时间：早上和晚上。一开始，代表一个动物的结构体在给食（feeding）字段中包含 false。*feed-animal* 程序应当读入一个代表动物的结构体和一个给食时间，把动物（animal）结构体中相应的字段改变为 true。

接下来的两个例子是关于变化器的，其数据定义涉及到自我引用。如果要处理没有大小限制的数据，就需要使用自我引用。我们所遇到的第一种这样的数据是表，第二种是自然数。

以通讯录为例我们先来观察一下对结构体的表的修改。通讯录是条目表；为了完整起见，下面给出结构体和数据的定义：

```
(define-struct entry (name number))
```

> entry（条目）是结构体：
> (make-entry n p)
> 其中 n 是符号，p 是数。

> address-book（通讯录）是下列二者之一：
> 1. 空表，即 empty。
> 2. (cons an-e an-ab)，其中 an-e 是条目，an-ab 是通讯录。

只有第二个数据是自我引用的，所以我们把注意力集中于它的模板：

```
;; fun-for-ab : address-book -> XYZ
(define (fun-for-ab ab)
  (cond
    [(empty? ab) ...]
    [else ... (fun-for-entry (first ab)) ... (fun-for-ab (rest ab)) ...]))
```

如果需要用一个辅助函数来处理 *entry*，我们可能还需要写出处理结构体的函数的模板。

所以，假设我们需要一个更新现有条目的函数。这个函数读入一个 *address-book*，一个人名和一个电话号码。（通讯录中）第一个包含这个人名的 *entry* 要被修改，使之包含新的电话号码：

```
;; change-number! : symbol number address-book -> void
;; 效果：修改 ab 中第一个包含 name 的条目，
;; 使得它的 number 字段变为 phone，
(define (change-number! name phone ab) ...)
```

可以证明，用变化器来开发这个函数是有效的，因为当某一个条目改变时，通讯录中大多数的条目保持不变。下面是一个例子：

```
(define ab
  (list
    (make-entry 'Adam 1)
    (make-entry 'Chris 3)
    (make-entry 'Eve 2)))
```

```
(begin
  (change-number! 'Chris 17 ab)
  (= (entry-number (second ab)) 17))
```

这个定义引入了 *ab*，一个有三个条目的 *address-book*。begin 表达式先修改 *ab*，把'Chris 与 17 相关连；接着它比较 *ab* 中第二个条目的号码与 17。如果 *change-number!*正确运行，begin 表达式的返回值会是 true。一个更好的测试还应确保 *ab* 中的其他东西都没有被改变。

下一步就是使用模板和例子开发函数定义了。我们分别考虑每一种情况：

1.如果 *ab* 是空的，那么 *name* 并不在其中出现。不幸的是，用途说明并没有指定在这种情况下函数应该做些什么，而且这时函数确实没有什么明智的事可做。为了安全起见，我们使用 error 来发出一个错误消息，表示没有任何相应的条目被找到。

2.如果 *ab* 包含了一个条目（第一条），那么它可能包含 *name*，也可能不包含。要查明这一点，函数必须用一个 cond 表达式来区别两种情况：

```
(cond
  [(symbol=? (entry-name (first ab)) name) ...]
  [else ...])
```

在第一个子情况中，函数必须修改这个结构体。在第二个子情况中，*name* 可能会在(rest *ab*)中出现，这意味着函数必须修改表的其余部分中的某个 *entry*。所幸的是，自然递归正好完成这件事。

把所有这些东西放到一起，就得到了如下的定义：

```
(define (change-number! name phone ab)
  (cond
    [(empty? ab) (error 'change-number! "name not in list")]
    [else (cond
            [(symbol=? (entry-name (first ab)) name)
             (set-entry-number! (first ab) phone)]
            [else
             (change-number! name phone (rest ab))])]))
```

这个函数惟一独特的地方是，它在一种情况下使用了结构体的变化器。否则，它就是我们所熟悉的递归外形：有两个子句的 cond 以及一个自然递归。把这个函数与第 9.3 节中的 *contains-doll?*和习题 9.3.3 中的 *contains?*比较特别有指导意义。

习题

习题 41.2.9　定义 *test-change-numbe*，该函数读入一个人名、一个电话号码和一本通讯录。它使用 *change-number!*来修改通讯录，然后确保它被正确修改了。如果真的是这样，它就返回 true；如果不是这样，它产生一条错误消息。使用这个新的函数，至少用三个不同的例子对 *change-number!*进行测试。

习题 41.2.10　设计函数 *move-squares*，读入一个 *square* 表和数 *delta*，其中 *square* 表定义如前，函数修改表中每一个正方形，把 *delta* 加到它的位置的 *x* 分量上。

习题 41.2.11　开发函数 *all-fed*，它读入一个 *animal* 的表，其定义如习题 41.2.8，函数修改这些动物，使它们早上的给食字段改变为 true。

习题 41.2.12　开发函数 *for-all*，该函数对习题 41.2.10 和习题 41.2.11 中的 *move-squares* 和 *all-fed* 进行抽象。函数读入两个值：一个读入结构体并返回(void)的函数以及一个结构体的表，函数的返回值是(void)。

习题 41.2.13　设计函数 *ft-descendants*，它读入一个基于如下结构体的后代家谱树（参见第 15.1 节）：

```
(define-struct parent (children name date eyes no-descendants))
```

在 *parent* 结构体中，最后的字段（后代的数量）最初是 0。函数 *ft-descendants* 遍历树，并修改这些位置，使得它们包含相应家族成员的后代总数。函数的返回值是给定的树的后代总数。

自然数是另一类需要自我引用描述的值。本质上，对自然数的递归与变化的结合并不太有用，但是，当数据表示涉及到向量时，对作为向量下标的自然数进行递归就很有用。

我们从一个电梯控制程序片断开始。电梯控制程序必须知道人们在哪些楼层按了呼叫按钮。我们假设电梯的硬件可以改变某些布尔值的状态，更确切地说，假设这个程序包含一个称为 *call-status*（呼叫状态）的向量，如果有人按了某个楼层的呼叫按钮，*call-status* 中相应的字段就是 true。

一个非常重要的电梯操作是 *reset*（重设）所有的按钮。例如，当电梯暂时不能正常工作时，操作员可能不得不重新启动电梯。我们通过重新叙述在 Scheme 框架中已知的事实开始来开发 *reset*[1]：

```
;; call-status : (vectorof boolean)
;; 记录哪些楼层发出了呼叫
(define call-status (vector true true true false true true true false))

;; reset : -> void
;; 效果：把 call-status 中所有的字段设为 false
(define (reset) ...)
```

前一个定义指定 *call-status* 为一个状态变量，但是，我们当然把向量中的每一个位置当作状态变量来使用，而不是把整个向量当状态变量使用。后一个定义由三个部分组成：合约、效果说明与函数 *reset* 的头部，它们实现了服务的非正式说明。

虽然我们可以把这项服务实现为：

```
(define (reset)
  (set! call-status
    (build-vector (vector-length call-status) (lambda (i) false))))
```

但是这种平凡的解决方案显然不是我们所想要的，因为它建立了一个新的向量。与此不同，我们需要一个函数，它修改现有向量中的每一个字段。遵循第 29 章中的提议，我们使用如下模板开发一个辅助函数：

```
;; reset-aux : (vectorof boolean) N -> void
;; 效果：把 v 中下标在[0, i)之间的字段设为 false
(define (reset-aux v i)
  (cond
    [(zero? i) ...]
    [else ... (reset-aux v (sub1 i)) ...]))
```

也就是说，这个辅助函数不仅仅读入向量，还读入一个限定区间。这个模板的形状基于后一个数据定义的。

函数的效果说明表明下列例子：

1．(*reset-aux call-status* 0)保持 *call-status* 不变，因为用途说明它要修改在[0,0）之间的下标，而这之间没有下标；

2．(*reset-aux* 1)改变 *call-status*，使得(vector-ref *call-status* 0)变为 false，因为在[0, 1）之间惟一的自然数就是 0；

3．(*reset-aux call-status* (vector-length *call-status*))把 *call-status* 中所有的字段设为 false。

其中最后一个例子提示我们使用(*reset-aux call-status* (vector-length *call-status*))来定义 *reset*。

有了这些例子，我们可以把注意力集中到定义之上了。关键是要记住额外的参数被解释为向量的下

[1] 记号(**vectorof** *X*)类似于(**listof** *X*)。

标。记住例子和方针，下面分别来观察两种情况：

1. 如果(zero? *i*)成立，那么函数没有任何效果，并且返回(void)。

2. 否则 *i* 就是正的。在这种情况下，自然递归会把 *call-status* 中所有下标在[0,(sub1 *i*)）之间的字段设为 false。另外，要完成整个任务，函数必须把向量中下标为(sub1 *i*)的字段设为 false。我们用一个 begin 表达式列出自然递归和另外的 vector-set!，从而把两个效果结合起来。

图 41.1 把所有这些东西集中到了一起。*reset-aux* 定义中的第二个子句修改向量中下标为(sub1 *i*)的字段，然后使用自然递归。函数的返回值就是 begin 表达式的返回值。

```
;; call-status : (vectorof boolean)
;; 记录哪些楼层发出了呼叫
(define call-status (vector true true true false true true true false))

;; reset : -> void
;; 效果：把 call-status 中所有的字段设为 false
(define (reset)
  (reset-aux call-status (vector-length call-status)))

;; reset-aux : (vectorof boolean) N -> void
;; 效果：把 v 中下标在[0, i)之间的字段设为 false
(define (reset-aux v i)
  (cond
    [(zero? i) (void)]
    [else (begin
              (vector-set! v (sub1 i) false)
              (reset-aux v (sub1 i)))]))
```

图 41.1 重设电梯的呼叫按钮

习题

习题 41.2.14 使用一些例子作为 *reset-aux* 的测试样板。把测试表示为布尔值表达式。

习题 41.2.15 开发 *reset* 如下的变体：

```
;; reset-interval : N N -> void
;; 效果：把[from, to]中所有的字段设为 false
;; 假设：(<= from to)成立
(define (reset-interval from to) ...)
```

使用 reset-interval 定义 reset。

习题 41.2.16 假设我们用一个向量来表示某个物体的位置，用另一个向量来表示它的速度。开发函数 *move!*，该函数读入一个位置向量和一个长度相等的速度向量，函数修改位置向量，一个字段一个字段地加上速度向量中的数值：

```
;; move! : (vectorof number) (vectorof number) -> void
;; 效果：把 v 中的字段加到相应的 pos 字段上
;;
;; 假设：pos 和 v 的长度相等
(define (move! pos v) ...)
```

证明使用这个修改向量的函数适合于模拟物体的运动。

习题 41.2.17 开发函数 *vec-for-all*，该函数抽象 *reset-aux* 和习题 41.2.16 中 *move!*的辅助向量处理函数。这个函数读入两个值：函数 *f* 和向量 *vec*。函数 f 读入下标（*N*）和向量的元素。*vec-for-all* 的返回值是(void)：它的效果是把 *f* 作用于 *vec* 中所有的下标和相应的值：

```
;; vec-for-all : (N X -> void) (vectorof X) -> void
;; 效果：把 f 作用于 vec 中所有的下标和值
;; 等式：
;; (vec-for-all f (vector v-0 ... v-N))
;; =
;; (begin (f N v-N) ... (f 0 v-0) (void))
(define (vec-for-all f vec) ...)
```

使用 *vec-for-all* 定义 *vector*!*，这个函数读入数 *s* 和一个数向量，修改这个向量，用 *s* 去乘其中的每个字段。

最后一个例子覆盖了一种一般的情况，即我们想要一次计算多个数的值，并把它们放在一个向量中。在第 37 章中，我们看到，有时候使用效果对一次传送多个结果来说是有用的。以同样的方式，有时候我们最好在同一个函数内创建向量并对它进行修改。考虑如下的问题，计算在一个字母表中每个元音字母出现了多少次：

```
;; count-vowels : (listof letter)
;;            -> (vector number number number number number)
;; 其中 letter 是一个在'a……'z 之中的符号
;; 判断五个元音字母在 chars 中出现了几次
;; 返回的向量按照字母顺序列出计得的总数
(define (count-vowels chars) ...)
```

选用向量作为返回值是正确的，因为这个函数必须把五个值结合到一个值中，而这五个值的重要性是相等的。

使用用途说明还可以得出一些例子：

```
  (count-vowels '(a b c d e f g h i))
= (vector 1 1 1 0 0)
  (count-vowels '(a a i u u))
= (vector 2 0 1 0 2)
```

既然输入是一个表，自然我们会选择表处理函数的模板：

```
(define (count-vowels chars)
  (cond
    [(empty? chars) ...]
[else ... (first chars) ... (count-vowels (rest chars)) ... ]))
```

要填写这个模板中的空缺，分别考虑两个子句：

1. 如果(empty? *chars*)为 true，那么返回值是一个向量，其中包含了五个 0。毕竟，在空表中没有元音字母。

2. 如果 *chars* 不是 empty，那么自然递归会计算出在(rest *chars*)中出现的元音字母的数量。要得出正确的结果，我们还需要检查(first *chars*)是不是元音，再根据这个结果，对相应的字段加一。既然这项任务是一个独立的、重复的任务，我们把它交给辅助函数处理：

```
;; count-a-vowel : letter
;;       (vector number number number number number) -> void
;; 效果：如果 l 是一个元音字母，就修改相应位置的 counts
(define (count-a-vowel l counts)
  ...)
```

换句话说，第二个子句先计算表的其余部分中的元音字母的数量。按照用途说明，这个计算保证能返回一个向量。我们把这个向量称为 *counts*。接着，该子句使用 *count-a-vowel* 来增加 *counts* 中相应字段的值——如果需要增加的话。函数的返回值是 *counts*，这是在表的第一个字母已被计数后的值。

```
;; count-vowels : (listof letter)
;;         -> (vector number number number number number)
;; 其中 letter 是一个在'a……'z 之中的符号
;; 判断五个元音字母在 chars 中出现了几次
;; 返回的向量按照字母顺序列出计得的总数
(define (count-vowels chars)
  (cond
    [(empty? chars) (vector 0 0 0 0 0)]
    [else
     (local ((define count-rest (count-vowels (rest chars))))
       (begin
         (count-a-vowel (first chars) count-rest)
         count-rest))]))

;; count-a-vowel : letter (vector number number number number number) -> void
;; 效果：如果 l 是一个元音字母，就修改相应位置的 counts，
;; 否则就不做任何事
(define (count-a-vowel l counts)
  ...)
```

图 41.2　计算元音字母的数量

图 41.2 给出了主函数的完整定义。辅助函数的定义完全按照非递归结构体改变的诀窍进行。

习题

习题 41.2.18　开发函数 *count-a-vowel*，然后测试完整的 *count-vowels* 程序。

习题 41.2.19　在第 29 章的最后，我们就已经可以定义了如图 41.3 所示的 *count-vowels* 了。这个版本的函数并没有使用 vector-set!，而是直接使用 build-vector 来构造向量。

测量 *count-vowels-bv* 与 *count-vowels* 的性能差别。提示：定义一个函数，生成一个巨大的随机字母表（比方说，有 5000 或 10000 个元素）。

解释 *count-vowels-bv* 与 *count-vowels* 之间的性能差别。这能不能解释测量到的差别？对 vector-set! 操作来说，这表明了什么？

习题 41.2.20　设计函数 *histogram*，该函数读入一个成绩表，表中的每个成绩都在 0 和 100 之间；它返回一个长度为 101 的向量，其中的每一个位置包含这个分数出现的次数。

习题 41.2.21　设计 *count-children*，该函数读入一棵后代家谱树，即从一个家庭成员指向其后代的家谱树。它返回一个有六个字段的向量。第一个字段包含没有子女的家庭成员的数目；第二个字段包含只有一个子女的家庭成员的数目；第三个字段包含只有两个子女的家庭成员的数目；……；最后一个字段包含有五个或更多子女的家庭成员的数目。

```
(define (count-vowels-bv chars)
  (local ((define (count-vowel x chars)
            (cond
              [(empty? chars) 0]
              [else (cond
                      [(symbol=? x (first chars))
                       (+ (count-vowel x (rest chars)) 1)]
                      [else (count-vowel x (rest chars))])])))
    (build-vector 5 (lambda (i)
                      (cond
                        [(= i 0) (count-vowel 'a chars)]
                        [(= i 1) (count-vowel 'e chars)]
                        [(= i 2) (count-vowel 'i chars)]
                        [(= i 3) (count-vowel 'o chars)]
                        [(= i 4) (count-vowel 'u chars)])))))
```

图 41.3　另一种对元音字母计数的方法

41.3　结构体的设计诀窍与变化器之二

在前面的章节中，我们学习了字段中包含原子值的结构体的变化器。然而，结构体还可以包含结构体。从第 14.1 节开始，我们还遇到了自我引用的数据定义，其中涉及到了结构体中的结构体。有时候，处理这种类型的数据可能也需要相应的、包含结构体的结构体字段的变化器。这一节就讨论一个这样的例子。

假设我们想要用一个程序来模拟纸牌游戏。每一张牌都有两个重要的属性：花色（suit）和等级（rank）。游戏者手中的牌被称为 hand。我们暂时还假设每一位游戏者手中至少有一张牌，也就是说，hand 永远不是空的。

图 41.4 给出了一个纸牌游戏的屏幕截图，其中包含了用来操作纸牌和 hand 的 DrScheme 结构体和数据定义。这个程序片断并没有引入独立的纸牌和 hand 定义，而是给出一个结构体和 hand 的数据定义。hand 是由 *hand* 结构体组成的，这个结构体中包含 *rank*、*suit* 和 *next* 字段。这个数据定义表明 *next* 字段可以包含两种类型的值：empty 和一个 *hand* 结构体，empty 表示没有其他的牌了，而 *hand* 中则包含了其余的牌。从全局的观点看，*hand* 形成一个纸牌的链；只有最后一张牌的 *next* 字段中包含 empty[1]。

起初，游戏者手中没有牌。获得第一张牌就建立起一个 hand。接下来，其他的牌按需要被插入到现有的 hand 中。这需要两个函数：一个用来创建 hand，另一个用来向 hand 中插入一张牌。因为 hand 只存在一次，而且它对应于一个物理对象，所以我们很自然地把第二个函数看作是修改现有值的函数，而不是创建一个新值的函数。眼下，我们接受这个假定，并探测它的结果。

创建 hand 是一个简单的行为，很容易实现为一个函数：

```
;; create-hand : rank suit -> hand
;; 用 r 和 s 创建一个只有一张牌的 hand
(define (create-hand r s)
  (make-hand r s empty))
```

这个函数读入一张牌的属性；它创建一个 hand，其中只有一张牌。

把一张牌添加到 hand 的末端是一件更为困难的任务。稍微简化一下，我们假定游戏者总是把新的牌

[1] Scheme 提供有表的变化器，Scheme 程序设计者应该使用表的变化器把 *hand* 表示为由牌组成的表。

添加到 hand 的末端。在这种情况下，我们必须处理一个任意长的值，这意味着我们需要一个递归函数。下面是它的合约、效果说明和头部：

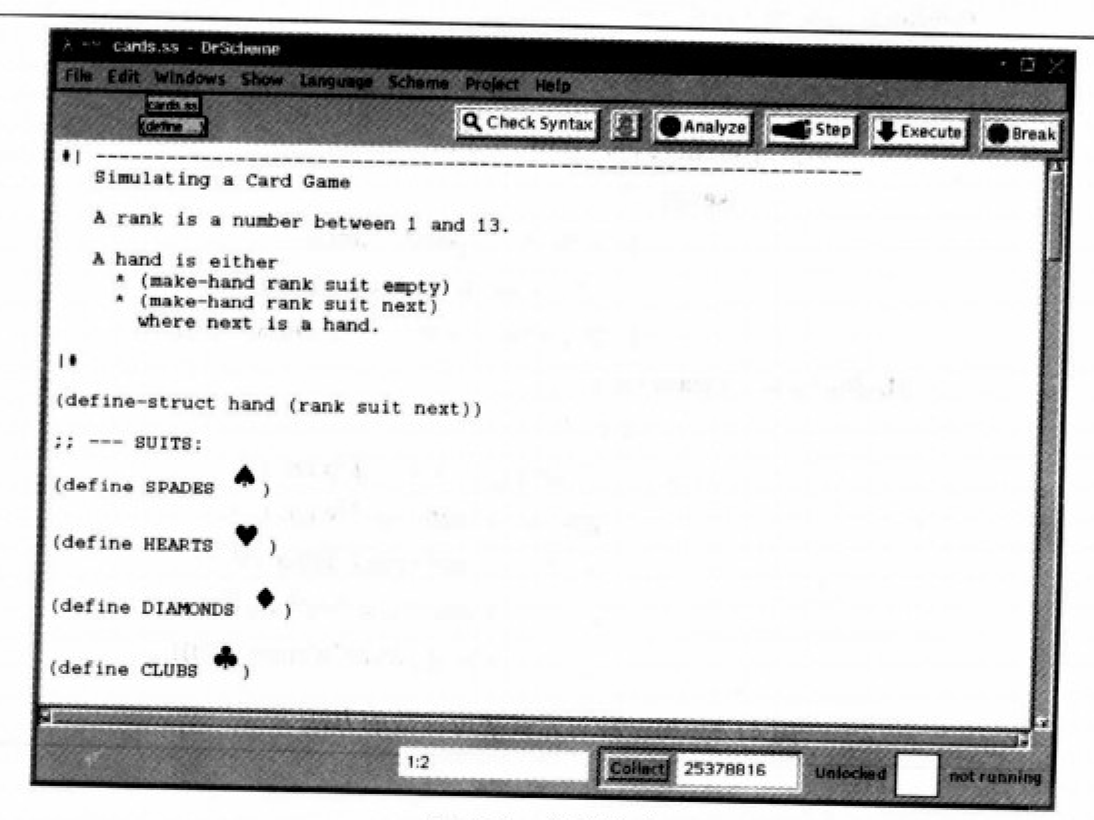


图 41.4　纸牌游戏

```
;; add-at-end! : rank suit hand -> void
;; 效果：在 a-hand 的尾部添加一张牌，其等级为 r，花色为 s
(define (add-at-end! rank suit a-hand) ...)
```

这些说明表明这个函数以不可见的值为返回值，而通过它的效果与其余的程序通信。

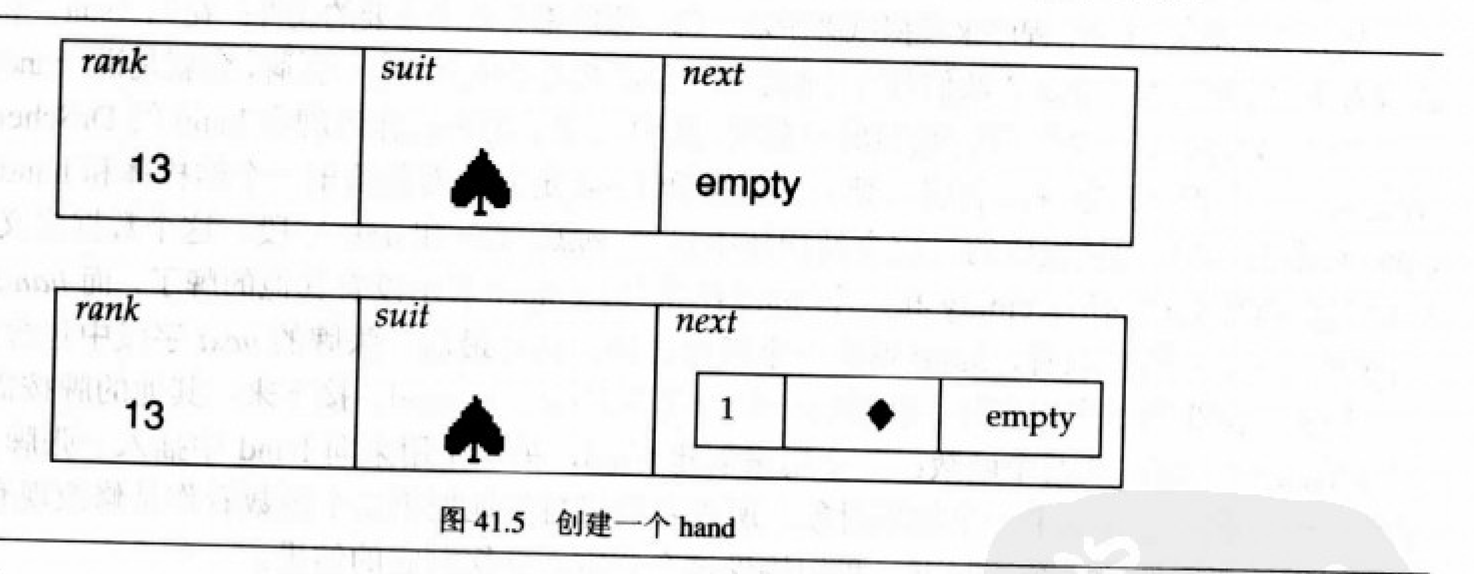


图 41.5　创建一个 hand

从例子开始：

```
(define hand0 (create-hand 13 SPADES))
```

如果在这个定义的环境下计算如下的表达式：

```
(add-at-end! 1 DIAMONDS hand0)
```

那么 *hand0* 就变成有两张牌的 hand：黑桃 13 后跟着方块 1。图 41.5 描绘了 *hand0* 的改变：前一半图显示了 *hand0* 的初始状态，后一半图显示了在 *add-at-end!*添加一张牌以后的状态。如果我们在这个环境下进一步计算

```
(add-at-end! 2 CLUBS hand0))
```

hand0 就会变成有三张牌的 hand。最后一张牌是梅花 2。用计算的话来说，执行了两个添加之后，*hand0* 的定义应当变为：

```
(define hand0
  (make-hand 13 SPADES
    (make-hand 1 DIAMONDS
      (make-hand 2 CLUBS empty))))
```

既然 *add-at-end!*的参数 *rank* 和 *suit* 是原子值，那么函数的模板必定是基于 *hand* 的数据定义的：

```
(define (add-at-end! rank suit a-hand)
  (cond
    [(empty? (hand-next a-hand))
     ... (hand-rank a-hand) ... (hand-suit a-hand) ...]
    [else ... (hand-rank a-hand) ... (hand-suit a-hand) ...
     ... (add-at-end! rank suit (hand-next a-hand)) ...]))
```

这个模板由两个子句组成，它们检查 *a-hand* 的 *next* 字段的内容。函数在第二个子句中是递归的，因为 *hand* 的数据定义在该子句中也是递归的。简而言之，这个模板完全是传统的。

下一个步骤是要考虑函数在每一个子句中应该怎样影响 *a-hand*：

1. 在第一种情况下，*a-hand* 的 *next* 字段为 empty。在这种情况下，我们可以修改这个 *next* 字段，使得它包含新牌：

```
(set-next-hand! a-hand (make-hand rank suit empty))
```

这个新建立的 *hand* 结构体现在在它的 *next* 字段中包含了 empty，也就是说，它是 *a-hand* 值新的尾部。

2. 在第二种情况下，自然递归会把新牌加到 *a-hand* 的尾部。事实上，因为给定的 *a-hand* 不是链中的最后一个，所以自然递归就完成了所需要做的所有事情。

下面是 *add-at-end!*的完整定义：

```
;; add-at-end! : rank suit hand -> void
;; 效果：在 a-hand 的尾部添加一张牌，其等级为 r，花色为 s
(define (add-at-end! rank suit a-hand)
  (cond
    [(empty? (hand-next a-hand))
     (set-hand-next! a-hand  (make-hand rank suit empty))]
    [else (add-at-end! rank suit (hand-next a-hand))]))
```

这个定义很类似于本书第二部分中设计的表处理函数。这一点应该毫不令人奇怪，因为 *add-at-end!* 处理的值所属的类型非常接近于表的数据定义，而设计诀窍是以一种一般的方式得出的。

习题

习题 41.3.1　手工计算如下的程序：

```
(define hand0 (create-hand 13 SPADES))

(begin
  (add-at-end! 1 DIAMONDS hand0)
  (add-at-end! 2 CLUBS hand0)
  hand0)
```

用这个例子对函数进行测试。

另外构造两个例子。回忆每个例子的组成成分：一个初始的 hand、要添加的牌以及对结果的预言。接下来用这些额外的例子测试函数。用布尔值表达式表示测试。

习题 41.3.2 开发函数 *last-card*。这个函数读入一个 *hand*，返回由最后一张牌的等级和花色组成的表。我们可以怎样用这个函数来测试 *add-at-end!*函数？

习题 41.3.3 假设家谱树是结构体组成的，结构体中记录了某个人的姓名、社会保障号码以及其父母。描述这样一个树需要有结构体的定义：

```
(define-struct child (name social father mother))
```

和数据定义：

> 家谱树节点（*family-tree-node*，简称 *ftn*）是下列两者之一：
> 1. *False*。
> 2. (make-*child name socsec f m*)，其中 *name* 是符号，*socsec* 是数，而 *f* 和 *m* 是 *ftn*。

暂时假设每个人都有一个社会保障号码，而且社会保障号码是惟一的。

遵照本书第三部分中的传统，false 代表缺少某个人的父亲或母亲的知识。不过，随着我们找出更多的信息，可以在家谱树中添加节点。

开发函数 *add-ftn!*。这个函数读入家谱树 *a-ft*、社会保障号码 *ssc*、符号 *anc* 以及一个 *child* 结构体。它的效果是修改 *a-ft* 中社会保障号码为 *ssc* 的结构体。如果 *anc* 是'father，函数就修改 *father* 字段，使之包含给定的 *child* 结构体；否则，*anc* 就是符号'mother，而 *add-ftn!*改变 *mother* 字段。如果相应的字段已经包含了一个 *child* 结构体，*add-ftn!*就产生一个错误消息。

用函数作参数： 这个函数除了可以接受'father 和'mother 作为 *anc* 以外，还可以接受两个结构体的改变器作参数：*set-child-father!*或者 *set-child-mother!*。按此相应修改 *add-ftn!*。

习题 41.3.4 使用封装的状态变量与函数的定义，开发一个 hand 的实现，其中包括 *create-hand* 和 *add-at-end!*服务。请使用 set!表达式，不要使用结构体的变化器。

并不是所有的变化器函数都像 *add-at-end!*一样容易开发。事实上，在某些情况下，程序甚至完全无法计算。我们来考虑两个额外的服务。第一项服务移去 hand 中的最后一张牌，它的合约与用途说明只需修改 *add-at-end!*的合约与用途说明：

```
;; remove-last! : hand -> void
;; 效果：移去 a-hand 中最后一张牌，除非 a-hand 中只有一张牌
(define (remove-last! a-hand) ...)
```

这个效果受到了限制，因为 hand 中必须包含一张牌。

我们还可以毫无困难地修改 *add-at-end!*的例子：

```
(define hand0 (create-hand 13 SPADES))

(begin
  (add-at-end! 1 DIAMONDS hand0)
  (add-at-end! 2 CLUBS hand0)
  (remove-last! hand0)
  (remove-last! hand0))
```

这个程序的返回值是 *void*。计算的效果是把 *hand0* 变回它的初始值。

*remove-last!*的模板与 *add-at-end!*的相同，因为这两个函数处理同一种类型的值。所以，下一步我们要分析这个函数在模板的每一种情况下所必须计算的效果是什么：

1. 回忆第一个子句所代表的情况，*a-hand* 的 *next* 字段为 empty。与 *add-at-end!*的情形不同，在这种情况下我们要做什么还不明确。依照用途说明，我们必须做这两件事中的一件：

a、如果 *a-hand* 是某个链的最后一个元素，而且这个链包含了超过一个的 *hand* 结构体，那么 *a-hand* 就要被移去。

b、如果 *a-hand* 是 *remove-last!*读入的惟一一个结构体，那么函数应该没有效果。

2. 但是我们无法获知 *a-hand* 是某条 *hand* 长链的最后一个，还是惟一的一个。在开始计算的时候，我们就丢失了这个本来可以获得的知识。

对第一个子句的分析使我们想到了使用累积器。我们尝试了自然的开发途径，发现在计算中有知识被丢失了，而这是考虑转而使用基于累积器的设计诀窍的评判标准。

一旦意识到需要使用积累类型的函数，就要把模板封装到一个 local 表达式中，并在它的定义和调用中添加一个累积器：

```
(define (remove-last! a-hand0)
  (local (;; accumulator ...
      (define (rem! a-hand accumulator)
        (cond
          [(empty? (hand-next a-hand))
           ... (hand-rank a-hand) ... (hand-suit a-hand) ...]
          [else ... (hand-rank a-hand) ... (hand-suit a-hand) ...
           ... (rem! (hand-next a-hand) ... accumulator ...) ...])))
    ... (rem! a-hand0 ...) ...))
```

现在的问题是，这个累积器代表了什么，它的初始值又是什么。

对 *remove-last!*来说，理解累积器本质最好的方法是，研究为什么普通的结构设计方法失效了。因此，我们回过头分析模板中的第一个子句。当 *rem!*运行到该子句的时候，有两件事应该已经完成了。第一件事，*rem!*应该知道 *a-hand* 不是 *a-hand0* 中惟一的 *hand* 结构体。第二件事，*rem!*必须要能够把 *a-hand* 从 *a-hand0* 中移去。要实现第一个目标，*rem!*的第一个调用应该在这样一个环境中，我们知道 *a-hand0* 包含了超过一张的牌。这一点说明在 local 表达式中要有一个 cond 表达式：

```
(cond
  [(empty? (hand-next a-hand)) (void)]
  [else (rem! a-hand0 ...)])
```

要实现第二个目标，*rem!*的累积器参数应该总是代表 *a-hand0* 中在 *a-hand* 前的 *hand* 结构体。这样，*rem!*通过修改前者的 next 字段就可以移去 *a-hand* 了：

```
(set-hand-next! accumulator empty)
```

现在，这个设计难题的各个部分变得清清楚楚了。函数的完整定义在图 41.6 中给出。*accumulator* 参数被改名为 *predecessor-of:a-hand*，用以强调它和严格意义上参数的关系。在 local 表达式中第一个对 *rem!*的调用把 *a-hand0* 中的第二个 *hand* 结构体传递给它。函数的第二个参数是 *a-hand0*，这样就建立了所需的关系。

现在到了重新访问纸牌游戏的基本假设的时候了，这个假设是：牌总是被加到 hand 的尾部。当人们拿到一张牌的时候，他们几乎从不把它放到 hand 的尾部，而是使用某种特殊的排列，并且在游戏的过程中保持这种排列不变。有的排列方法是按照花色排，有的排列方法是按照等级排，还有的方法同时使用这两种标准。

我们来考虑这样一种操作，它按照牌的等级把牌插入到某个 hand 中。

```
;; sorted-insert! : rank suit hand -> void
;; 假设：a-hand 是按照等级递减排列的
;; 效果：把一张等级为 r、花色为 s 的牌插入到合适的位置中
(define (sorted-insert! r s a-hand) ...)
```

这个函数假设给定的 hand 已经排好序了。如果总是使用 *create-hand* 来建立 hand，并且使用 *sorted-insert!*来插入牌，那么这个假设自然就会成立。

```
;; remove-last! : hand -> void
;; 效果，移去 a-hand0 中的最后一张牌，除非只剩一张牌了
(define (remove-last! a-hand0)
  (local (;; predecessor-of:a-hand 表示 a-hand 链中 a-hand 的前继
          (define (rem! a-hand predecessor-of:a-hand)
            (cond
              [(empty? (hand-next a-hand))
               (set-hand-next! predecessor-of:a-hand empty)]
              [else (rem! (hand-next a-hand) a-hand)])))
    (cond
      [(empty? (hand-next a-hand0)) (void)]
      [else (rem! (hand-next a-hand0) a-hand0)])))
两个 rem!调用的外形都是：
(rem! (hand-next a-hand) a-hand)
```

图 41.6　移去最后一张牌

假设我们按照与前述一样的例子开始使用 *add-at-end!*：

```
(define hand0 (create-hand 13 SPADES))
```

如果我们在这个环境下计算(sorted-insert! 1 DIAMONDS hand0)，hands0 就会变成：

```
(make-hand 13 SPADES
  (make-hand 1 DIAMONDS empty))
```

如果我们现在又计算(*sorted-insert!* 2 *CLUBS hand0*)，得到的 *hand0* 是

```
(make-hand 13 SPADES
  (make-hand 2 CLUBS
    (make-hand 1 DIAMONDS empty)))
```

这个值说明了按照降序排列的链的意义。随着我们从前向后遍历这个链，等级就会越来越小，而不论花色是什么。

下一个步骤是分析模板。下面是适应当前用途的模板：

```
(define (sorted-insert! r s a-hand)
  (cond
    [(empty? (hand-next a-hand))
     ... (hand-rank a-hand) ... (hand-suit a-hand) ...]
    [else ... (hand-rank a-hand) ... (hand-suit a-hand) ...
      ... (sorted-insert! r s (hand-next a-hand)) ...]))
```

关键的步骤是要把新牌插入到一个位置中，使得它的前一张牌的等级要比 *r* 大，或者一样大，而且 *r* 要比它的后一张牌的等级大，或者一样大。因为在第二个子句中只有两张牌，所以我们先给出第二个子句的答案。刚才所描述的条件表明，我们需要一个嵌套的 cond 表达式：

```
(cond
  [(>= (hand-rank a-hand) r (hand-rank (hand-next a-hand))) ...]
  [else ...])
```

第一个条件用 Scheme 表达了我们刚才所讨论的条件。具体来说，(*hand-rank a-hand*)选出 *a-hand* 中第一张牌的等级，(*hand-rank* (*hand-next a-hand*))选出第二张牌的等级。比较表达式判断这三个等级是否

按照正确的顺序排列。

这个新 cond 表达式的每一种情况都需要单独进行分析：

1. 如果(>= (*hand-rank a-hand*) r (*hand-rank* (*hand-next a-hand*)))为真，那么新的牌就必须进入当前所连接两张牌之间。也就是说，*a-hand* 的 *next* 字段必须修改，使之包含新的 *hand* 结构体。这个新的结构体由 *r*、*s* 和原来 *a-hand* 的 *next* 字段中的值组成。这就产生了如下的 cond 表达式细节：

```
(cond
  [(>= (hand-rank a-hand) r (hand-rank (hand-next a-hand)))
   (set-hand-next! a-hand (make-hand r s (hand-next a-hand)))]
  [else ...])
```

2. 如果(>= (*hand-rank a-hand*) r (*hand-rank* (*hand-next a-hand*)))为假，那么新的牌就必须被插入到链的其余部分中的某个位置。当然，自然递归正好完成这件事，这就结束了对 *sorted-insert!*的第二个子句的分析。

把所有的片断结合起来，就得到了部分函数定义：

```
(define (sorted-insert! r s a-hand)
  (cond
    [(empty? (hand-next a-hand))
     ... (hand-rank a-hand) ... (hand-suit a-hand) ...]
    [else
      (cond
    [(>= (hand-rank a-hand) r (hand-rank (hand-next a-hand)))
     (set-hand-next! a-hand (make-hand r s (hand-next a-hand)))]
    [else (sorted-insert! r s (hand-next a-hand))])]))
```

现在，惟一剩下的空缺就在第一个子句中。

第一个子句与第二个子句之间的区别是，在第一个子句中没有第二个 *hand* 结构体，所以我们不能对等级进行比较。不过，我们可以比较 *r* 和(*hand-rank a-hand*)，从而根据这个比较的结果计算出一些东西：

```
(cond
  [(>= (hand-rank a-hand) r) ...]
  [else ...])
```

显然，如果这个比较计算的结果是 true，那么函数必须修改 *a-hand* 的 *next* 字段，从而添加上新的 *hand* 结构体：

```
(cond
  [(>= (hand-rank a-hand) r)
   (set-hand-next! a-hand (make-hand r s empty))]
  [else ...])
```

问题是，在第二个子句中，我们没有东西可以计算。如果 *r* 比 *a-hand* 的等级大，那么新的牌应该被插入到 *a-hand* 的前继和 *a-hand* 之间。但是这种情况会被第二个子句发现。这个表面上的矛盾表明第二个子句中的省略号部分只需要处理一种情况：

仅当 *sorted-insert!*读入的等级 *r* 比 *a-hand* 中所有 *rank* 字段的值都大时，省略号部分才会被计算。

在这种情况下，*a-hand* 根本不应该被改变。毕竟，这时修改 *a-hand* 或者任何一个它内部的 *hand* 结构体都不能产生一个降序排列的牌链。

乍看来，我们可以用一个 set!表达式来改变 *hand0* 的定义，从而解决这个问题：

```
(set! hand0 (make-hand r s hand0))
```

不过，一般而言这种修补方法并不能工作，因为我们不能假定必须要修改的变量定义是哪一个。既

然表达式只能对值进行抽象，而不能对变量抽象，我们就没有办法在这个 set!表达式中对 *hand0* 进行抽象。

我们被困住了，至少就上述的情形而言，我们无法定义出 *sorted-insert!*。不过，有一个补救办法。如果引入一个单独的变量，代表当前的 *hand* 结构体，那么就可以结合使用赋值和结构体的变化器，从而插入一张新牌。窍门是不要让程序的任何其他部分引用这个变量，更不要让它们修改这个变量。否则，如前所述，一个简单的 set!就不能运作了。换一种说法，对每个 *hand* 结构体我们都需要一个状态变量，而且要把它封装在一个 local 表达式中。

图 41.7 给出了完整的函数定义。它遵循了第 39 章中的模式。这个函数本身对应于 *create-hand*，不过这个新的 *create-hand* 函数并不返回一个结构体，而是返回一个管理器函数。在这里，管理器只能处理一条消息：'insert；所有其他的消息都会被拒绝。'insert 消息会先检查新的等级是不是比 *the-hand*（即隐含变量）的第一张牌大。如果新的等级更大，管理器就修改 *the-hand*；如果不更大，就调用 *insert-aux!*，这个函数现在假定新牌被插入到链的中部。

Hand 是一种界面：
1、**'insert** :: *rank suit* -> *void*

```
;; create-hand : rank suit -> hand
;; 由某一张牌的 rank 和 suit 建立一个 hand
(define (create-hand rank suit)
  (local ((define-struct hand (rank suit next))

          (define the-hand (make-hand rank suit empty))
          ;; insert-aux! : rank suit hand -> void
          ;; 假设：hand 已经按照等级降序排列了
          ;; 效果：在合适的位置添加一张牌，
          ;; 其等级为 r，花色为 s
          (define (insert-aux! r s a-hand)
            (cond
              [(empty? (hand-next a-hand))
               (set-hand-next! a-hand (make-hand r s empty))]
              [else (cond
                      [(>= (hand-rank a-hand)
                           r
                           (hand-rank (hand-next a-hand)))
                       (set-hand-next! a-hand
                         (make-hand r s (hand-next a-hand)))]
                      [else (insert-aux! r s (hand-next a-hand))])]))

          ... ;; 其他所需的服务

          (define (service-manager msg)
            (cond
              [(symbol=? msg 'insert!)
               (lambda (r s)
                 (cond
                   [(> r (hand-rank the-hand))
                    (set! the-hand (make-hand r s the-hand))]
                   [else (insert-aux! r s the-hand)]))]
              [else (error 'managed-hand "message not understood")])))
    service-manager))
```

图 41.7 牌的 hands 的封装和结构体变化器

图 41.7　牌的 hands 的封装和结构体变化器

习题

习题 41.3.5　扩展图 41.7 中的定义，添加一项服务，移去第一张给定等级的牌，即使它是惟一的一张牌。

习题 41.3.6　扩展图 41.7 中的定义，添加一项服务，求出 *the-hand* 中某个给定等级的牌的花色。这个函数应该返回一个花色的表。

习题 41.3.7　重新给出图 41.7 中的 *create-hand*，使得管理器只使用一个 set!表达式，并且 *sorted-insert* 不再使用任何结构体的变化器。

习题 41.3.8　回忆第 14.2 节中二叉树的定义：

> 二叉树（*binary-tree*）(简称：*BT*)是下列两者之一：
> 1. false
> 2. (make-node *soc pn lft rgt*)，其中 *soc* 是数，*pn* 是符号，*lft* 和 *rgt* 是 *BT*。

所需的结构体定义是：

```
(define-struct node (ssn name left right))
```

一棵二叉树是二叉搜索树（binary-search-tree，简称 bst），如果每一个 *node* 结构体所包含的社会保障号码（*soc*）都比它的左子树（*lft*）中的社会保障号码大，而又比它的右子树（*rgt*）中的社会保障号码小。

开发函数 *insert-bst!*。这个函数读入人名 *n*、社会保障号码 *s* 以及一棵 *bst*。它修改该 *bst*，使得它包含一个新节点（该节点中包含 *n* 和 *s*），同时保持它为搜索树。

另外，开发函数 *remove-bst!*，该函数移去一个包含给定社会保障号码的节点。它合并被删除的节点的两棵子树，方法是把所有右子树中的节点插入到左子树中。

这一节与习题中的讨论说明，在链接结构（linked structures）中，添加或删除元素是一项棘手的任务。处理链接结构中间的元素最好是使用带累积器的函数。处理链接结构的第一个元素需要封装和管理函数。反之，如习题 41.3.7 所示，给出一个不带变化器的解决方案要比给出一个基于结构体的变化器的解决方案容易得多。而且，就牌和 hand 的问题而言，最多处理 52 个结构体，这两者在效率上是相等的。要决定使用两种方法中的哪一种，需要对算法分析（参见第 29 章）有更深刻的理解，还要对语言的机制和封装状态变量的设计诀窍有更深刻的理解。

41.4　补充练习：最后一次移动图片

在第 6.6 节、第 7.4 节、第 10.3 节和第 21.4 节中，我们学习了如何在画布上移动图片。图片是图形的表；图形是几个基本几何图形中的一种：圆、矩形等等。按照基本原则，每个概念一个函数，我们先定义移动基本几何图形的函数，接着定义移动混合图形的函数，最后定义移动图形的表的函数。最终，我们对相关的函数进行抽象。

移动基本图形的函数由一个现有的图形产生一个新的图形。例如，移动圆的函数读入一个 *circle* 结构体，返回一个新的 *circle* 结构体。如果我们把 *circle* 看作一个圆形框架的绘画，而把画布也看作绘画，那么，对每一个移动来说，建立一个新的图形就不合适了。作为代替，我们应该修改图形当前

的位置。

习题

习题 41.4.1 把习题 6.6.2 和习题 6.6.8 中的函数 *translate-circle* 和 *translate-rectangle* 分别改成结构体变化器的函数。修改第 6.6 节中的 *move-circle* 以及习题 6.6.12 中的 *move-rectangle*，让它们使用这些新的函数。

习题 41.4.2 使用习题 41.4.1 中结构体变化器的函数，修改习题 10.3.6 中的函数 *move-picture*。

习题 41.4.3 使用 Scheme 的 for-each 函数（参见 Help Desk）来抽象习题 41.4.2 中的函数，对所有可以抽象的地方进行抽象。

相等



在修改结构体或向量的时候，我们使用类似于“这个向量现在在它的第一个字段中包含 false”这样的词句来描述所发生的事。这些词句背后的含义是即使向量的属性发生了变化，向量自身仍然保持不变。这个观察说明，其实有两种相等的概念：一种是我们迄今为止所用的，另一种新的相等是基于结构体或向量的效果的。对程序设计者来说，理解这两种相等的概念是至关重要的。因此，接下来的两节中会详细讨论它们。

42.1 外延相等

回忆第一部分中的 *posn* 结构体类型。一个 *posn* 结合了两个数；它的两个字段被称为 *x* 和 *y*。下面是两个例子：

```
(make-posn 3 4)        (make-posn 8 6)
```

它们显然是不同的。反之，如下的两个 *posn*：

```
(make-posn 12 1)        (make-posn 12 1)
```

是相等的。它们在 *x* 字段中所包含的都是 12，在 *y* 字段中所包含的都是 1。

更一般地，如果两个结构体所包含的成分是相同的，我们就认为它们是相等的。这里假设我们知道怎样来比较其成分，但这并不使人吃惊。它正好提醒我们，处理结构体需要遵循数据定义，而与数据定义一起的还有结构体的定义。哲学家们把这种相等的概念称为外延的相等。

第 17.8 节介绍了外延相等，并讨论了它对于构造测试的作用。作为提示，我们来考虑一个函数，它判断 *posn* 结构体的外延相等性。

```
;; equal-posn: posn posn -> boolean
;; 判断两个 posn 外延上相等与否
(define (equal-posn p1 p2)
  (and (= (posn-x p1) (posn-x p2))
       (= (posn-y p1) (posn-y p2))))
```

这个函数读入两个 *posn* 结构体，提取出它们的字段值，然后用=比较相应的字段，其中的=是比较数的谓词。这个函数的组织形式与 *posn* 结构体的数据定义是相应的；它的设计是标准的。这表明，对于递归的数据类型，我们自然会需要递归的相等函数。

习题

习题 42.1.1 开发一个外延相等函数，比较习题 41.3.3 中的 *child* 结构体类型是否相等。如果 *ft1* 和 *ft2* 是两个家谱树节点，这个函数的最大抽象运行时间是多少？

习题 42.1.2 使用习题 42.1.1 对 *equal-posn* 进行抽象，使得这个函数的实例可以测试任何给定的

结构体的外延相等性。

42.2 内涵相等

考虑如下简单程序：

```
(define a (make-posn 12 1))
(define b (make-posn 12 1))

(begin
  (set-posn-x! a 1)
  (equal-posn a b))
```

该程序定义了两个 *posn* 结构体。在程序的前一部分，这两个结构体的起始值是相等的。但如果我们计算 begin 表达式，返回值是 false。

即使两个结构体一开始是由相同的值组成的，它们也是不同的，因为 begin 表达式中的结构体变化器修改了第一个结构体的 *x* 字段，而保持了第二个结构体不变。更一般地，这个表达式对第一个结构体有效果，而对第二个结构体没有效果。现在，来看一看一个稍有不同的程序：

```
(define a (make-posn 12 1))
(define b a)

(begin
  (set-posn-x! a 1)
  (equal-posn a b))
```

这里，*a* 和 *b* 是同一个结构体的两个名字。因此，对(*set-posn-x! a* 1)的计算同时影响了 *a* 和 *b*，这意味着这一次(*equal-posn a b*)会返回 true。

这两个观察有着一个一般的寓意。如果对某个表达式的计算会在影响一个结构体的同时影响另一个结构体，那么在某种意义上，这两个结构体的相等就要比 *equal-posn* 所能判断的相等更深刻。哲学家们把这种相等的概念称为内涵的相等。与外延的相等不同，这个相等的概念不仅仅需要两个结构体由相同的部分组成，还要求它们对结构体的变化器同时作出同样的响应。作为直接的结果，两个内涵相等的结构体也是外延相等的。

设计一个函数来判断结构体的内涵相等要比设计一个函数来判断结构体的外延相等复杂得多。必须从精确的描述开始：

```
;; eq-posn : posn posn -> boolean
;; 判断两个 posn 结构体是否对修改收到同样的影响
(define (eq-posn p1 p2) ...)
```

我们已经有了一个例子，所以接下来讨论模板：

```
(define (eq-posn p1 p2)
  ... (posn-x p1) ... (posn-x p2) ...
  ... (posn-y p1) ... (posn-y p2) ... )
```

这个模板中包含了四个表达式，每个表达式都告诉我们可用的信息是什么，我们能够修改的结构体又是什么。

把前面的观察翻译成完整的定义，就可以得到这样的函数草稿：

```
(define (eq-posn p1 p2)
  (begin
    (set-posn-x! p1 5)
    (= (posn-x p2) 5)))
```

这个函数把 *p1* 的 *x* 字段设置为 5，然后检查 *p2* 的 *x* 字段是不是也变成了 5。如果它也是 5，那么这两个结构体都对修改产生了响应，所以，按照定义，它们是内涵相等的。

不幸的是，这个推理中有一个问题。考虑如下的应用：

```
(eq-posn (make-posn 1 2) (make-posn 5 6))
```

这两个 *posn* 连外延都不相等，所以它们不可能是内涵相等的。但是，第一个版本的 *eq-posn* 会返回 true，这就是它的问题。

我们可以使用另一个修改器来改进第一个版本：

```
(define (eq-posn p1 p2)
  (begin
    (set-posn-x! p1 5)
    (set-posn-x! p2 6)
    (= (posn-x p1) 6)))
```

这个函数先修改 *p1*，再修改 *p2*。如果这两个结构体是内涵相等的，那么对 *p2* 的修改必然会影响到 *p1*。另外，我们知道 *p1* 的 *x* 字段不可能偶然地包含了 6，因为我们已经先把它改为 5 了。因而，如果(*eq-posn a b*)返回了 true，那么在 *a* 改变的同时 *b* 也会改变，反之亦然，所以这两个结构体是内涵相等的。

现在所剩下的惟一一个问题就是，*eq-posn* 对它所读入的两个结构体是有影响的，但是在它的说明中，并没有出现这种影响。事实上，函数不应该存在可见的影响，因为它惟一的用途是判断两个结构体是不是内涵相等。通过先把 *p1* 和 *p2* 原来的 *x* 字段值保存起来，然后修改字段，最后恢复原值，就可以避免这种影响。图 42.1 给出了一个函数定义，它可以执行一次内涵相等性的检查，同时不出现任何可见的效果。

```
;; eq-posn : posn posn -> Boolean
;; 判断两个 posn 结构体是否对修改收到同样的影响
(define (eq-posn p1 p2)
  (local (;; 保存 p1 和 p2 原来的 x 值
          (define old-x1 (posn-x p1))
          (define old-x2 (posn-x p2))
          ;; 修改 p1 的 x 字段以及 p2 的 x 字段
          (define effect1 (set-posn-x! p1 5))
          (define effect2 (set-posn-x! p2 6))
          ;; 现在比较这两个字段
          (define same (= (posn-x p1) (posn-x p2)))
          ;; 恢复原来的值
          (define effect3 (set-posn-x! p1 old-x1))
          (define effect4 (set-posn-x! p2 old-x2)))
    same))
```

图 42.1 判断两个结构体的内涵相等性

eq-posn 的存在表明所有的结构体都有一个独特的“指纹”。如果我们有权访问两个同一类型的结构体的修改器，就可以检查这个指纹。Scheme 和许多其他的语言一般都提供内建的函数，用来比较两个结构体值的外延相等与内涵相等。相应的 Scheme 函数是 equal?和 eq?。在 Scheme 中，这两个函数可以用于所有的值，无论它们的修改器与选择器是可用的还是隐含的。eq?的存在表明我们测试的原则需要有所

修改。

> **测试原则**
>
> 当比较对象的恒等性时，使用 eq?。对于其他的测试，使用 equal?。

这个方针是很普通的。不过，程序员仍然需要使用类似于 symbol=?、boolean?或者=的相等性函数，指出所需比较的值属于哪种类型，因为这些补充信息可以帮助读者更方便地理解程序的用途。

习题

习题 42.2.1 手工计算下列表达式：

```
1. (eq-posn (make-posn 1 2) (make-posn 1 2))

2. (local ((define p (make-posn 1 2)))
     (eq-posn p p))

3. (local ((define p (make-posn 1 2))
           (define a (list p)))
     (eq-posn (first a) p))
```

使用 DrScheme 检查答案。

习题 42.2.2 开发一个内涵相等性函数，用于习题 41.3.3 中的 *child* 结构体类型。如果 *ft1* 和 *ft2* 是家谱树节点，这个函数的最大抽象运行时间是多少？

习题 42.2.3 使用习题 42.2.2 对 *eq-posn* 进行抽象，使得这个函数的实例可以测试任何给定的结构体的内涵相等性。



修改结构体、向量和对象

第 43 章

本章介绍几个涉及可变结构体的小规模项目。其中小节的安排顺序大致与本书的大纲相一致，从简单的数据类型到复杂的，从结构递归到带回溯并使用累积器的生成递归。

43.1 关于向量的更多练习

前面的程序类型，几乎没有需要对可变向量进行编程。不过，可变向量在传统语言中非常普遍，所以对它进行编程是一种重要的技巧，除了 41.2 节中的练习之外，我们还需要更多练习。这一节讨论结构体的排序，但其目的是在处理向量时学习与区间有关的推理技巧。

早在 12.2 节设计 *sort* 函数时就遇到过排序算法，我们设计一个函数，该函数读入一个数表，返回一个包含相同数的表，其中所有数按照升序或降序进行了排列。一个类似的向量函数读入一个向量，返回一个新的向量。不过，使用向量修改器，我们也可以设计出一个函数，修改这个向量，使得它包含与原来一样的元素，而且是排好序的。这样一个函数被称为在原来的位置上排序，因为它把现有的向量中所有的元素保留在向量之中。

in-place-sort（在原来的位置上排序）函数只是依靠它对输入向量的效果来完成任务：

```
;; in-place-sort : (vectorof number) -> void
;; 效果：修改 V，使得它包含原来的元素，
;; 但是按照升序排列
(define (in-place-sort V) ...)
```

例子必须要能够说明效果：

```
(local ((define v1 (vector 7 3 0 4 1)))
  (begin
    (in-place-sort v1)
    (equal? v1 (vector 0 1 3 4 7))))
```

当然，既然 *in-place-sort* 读入一个向量，真正的问题是要设计一个辅助函数，处理向量的特定片断。标准的向量处理函数的模板使用了一个辅助函数：

```
(define (in-place-sort V)
  (local ((define (sort-aux V i)
        (cond
          [(zero? i) ...]
          [else
         ... (vector-ref V (sub1 i)) ...
```

```
          ... (sort-aux V (sub1 i)) ...])))
     (sort-aux V (vector-length V))))
```

遵照第 29 章中的设计思想，辅助函数读入一个自然数，把它当作向量的下标来使用。因为初始的参数是(vector-length *V*)，所以可以使用的下标总是(sub1 *i*)。

回忆一下，设计如 *sort-aux* 这样的函数的关键是给出严格的用途说明以及（或者）效果说明。这个说明必须阐明函数是在处理可能的向量下标中的哪个区间，并且准确地说明它完成了什么。一种自然的效果说明是这样的：

```
;; sort-aux : (vectorof number) N -> void
;; 效果：在原来的位置上对 V 的 [0, i) 区间排序
(define (sort-aux V i) ...)
```

要在更大的范围中理解这个效果说明，可以修改原来的例子：

```
(local ((define v1 (vector 7 3 0 4 1)))
  (begin
    (sort-aux v1 5)
    (equal? v1 (vector 0 1 3 4 7))))
```

如果 *sort-aux* 被作用于该向量的长度，那么它应该对整个向量排序。这个效果说明还表明，如果这个参数比向量的长度短，那么只有向量中起始的一段会被排序：

```
(local ((define v1 (vector 7 3 0 4 1)))
  (begin
    (sort-aux v1 4)
    (equal? v1 (vector 0 3 4 7 1))))
```

在这个特定的例子中，最后一个数仍被遗留在它原来的位置上，而向量的前四个元素已被排过序了。现在可以分析 *sort-aux* 模板中每一个子句了：

1.如果 *i* 是 0，那么效果说明中的区间就是[0，0]。这意味着这个区间是空的，所以函数不需要做任何事情。

2.模板中的第二个子句包含了两个表达式：

```
        (vector-ref V (sub1 i))
```

以及

```
        (sort-aux V (sub1 i))
```

第一个表达式提示我们可以使用 *V* 的第 i－1 个字段；第二个表达式提示我们可以使用自然递归。在这种情况下，自然递归对区间[0，(sub1 *i*)）进行排序。要完成任务，必须把第 i－1 个字段的值插入到它在区间[0，*i*）中合适的位置。

上述的例子可以更具体地描述这种情况。在计算(*sort-aux v1* 4)时，*v1* 最后一个字段中的数保持在原来的位置上。向量的前四个元素现在是：0，3，4 和 7。要对整个区间[0，5)排序，必须把 1，也就是(vector-ref *V* (sub1 5))，插入到 0 和 3 之间。

简而言之，到此为止，*in-place-sort* 的设计完全是按照 12.2 节中 *sort* 函数的模式进行的。对 *sort* 来说，在设计主函数时我们也发现还需要设计一个辅助函数，把某个元素插入到它合适的位置上。

图 43.1 整理了我们迄今为止对 *in-place-sort* 所进行的讨论，还给出了第二个辅助函数 *insert* 的说明。要理解这个函数的效果说明，我们再给出 *sort-aux* 的第二个子句的例子：

```
(local ((define v1 (vector 0 3 4 7 1)))
  (begin
    (insert 4 v1)
    (equal? v1 (vector 0 1 3 4 7))))
```

在这种情况下，*insert* 把 1 移过三个数：先是 7，接着是 4，最后是 3。当左侧的下一个数，也就是 0，比要插入的数还小时，函数就停止。

```
;; in-place-sort : (vectorof number) -> void
;; 效果：修改 V，使得它包含原来的元素，
;; 但是按照上升的顺序排列
(define (in-place-sort V)
  (local (;; sort-aux : (vectorof number) N -> void
                ;; 效果：在原来的位置上对 V 的[0，i]区间排序
                (define (sort-aux V i)
                  (cond
                    [(zero? i) (void)]
                    [else (begin
                            ;; 对片断[0，(sub1 i)]排序：
                            (sort-aux V (sub1 i))
                            ;; 把(vector-ref V (sub1 i))插入到片断
                            ;; [0，i] 中，使得它变为有序的
                            (insert (sub1 i) V))]))

          ;; insert : N (vectorof number) -> void
          ;; 把第 i 个位置上的值插入到 V 的片断
          ;; [0，i]中的合适位置上
          ;; 假设：V 的片断[0，i）已经被排好序
          (define (insert i V)
            ...))
    (sort-aux V (vector-length V))))
```

图 43.1　一个在原来位置上排序的函数：第一部分

再来看 *insert* 的第二个例子：

```
(local ((define v1 (vector 7 3 0 4 1)))
  (begin
    (insert 1 v1)
    (equal? v1 (vector 3 7 0 4 1))))
```

这里的问题是要把 3 插入到一个片断中，而这个片断只包含一个数：7。这意味着 *insert* 在交换第一和第二个字段之后必须停止，因为 3 不能再进一步地被左移了。

现在再来观察 *insert* 的模板：

```
(define (insert i V)
  (cond
    [(zero? i) ...]
    [else
      ... (vector-ref V (sub1 i)) ...
      ... (insert (sub1 i) V) ... ]))
```

这是标准的向量处理辅助函数模板。与往常一样，我们区分两种情况：

1. 如果 *i* 是 0，那么目标就是把(vector-ref *V* 0)插入到片断[0，0]之中。既然这个区间中只包含一个数，*insert* 已经完成了它的任务。

2. 如果 *i* 是一个正数，那么模板提示我们考虑 *V* 中的另一个名为(vector-ref *V* (sub1 *i*))的元素，并且可以进行一次自然递归。现在的问题是，(vector-ref *V* (sub1 *i*))是比(vector-ref *V* *i*) ——要被移动的元素—

一大还是小。如果是小，那么 *V* 在整个区间[0，i]上就已经是有序的了，因为按照假设，*V* 在区间[0，*i*]上已经是有序的了。如果是大，那么这个位于 *i* 的元素还不是在有序的位置上。

cond 表达式必须使用的条件是：

```
(cond
  [(> (vector-ref V (sub1 i)) (vector-ref V i)) ...]
  [(<= (vector-ref V (sub1 i)) (vector-ref V i)) (void)])
```

其中的第二个子句包含了(void)，因为这时不需要做任何的事。在第一个子句中，*insert* 必须（至少）交换两个字段的值。更确切地说，*insert* 必须把(vector-ref *V i*)放到字段(sub1 *i*)，把(vector-ref *V* (sub1 *i*)) 放到字段 i。但是，这还是不够的。毕竟，在第 *i* 个字段中的值可能会需要移过多个字段，就如同第一个例子所示的。万幸的是，我们可以使用自然递归轻松地解决这个问题，因为在完成交换之后，自然递归正好把(vector-ref *V* (sub1 *i*))插入到它在[0，(sub1 *i*)]中合适的位置。

```
(define (in-place-sort v)
```

图 43.2 一个在原来位置上排序的函数：第二部分

图 43.2 给出了 *insert* 和 *swap* 的完整定义。其中的第二个函数是用来交换两个字段的值的。

习题

习题 43.1.1 对图 43.1 和图 43.2 中 *in-place-sort* 的辅助函数进行测试。用布尔值表达式表示测试。设计 *in-place-sort* 的更多例子。

用这些部分构成一个完整的函数。测试这个完整的函数。一步一步地除去完整函数中辅助程序多余的参数，在每一个步骤之后都要测试完整的函数。最后，修改 *in-place-sort*，使得它的返回值是修改后的向量。

习题 43.1.2 图 43.2 中的 *insert* 函数在每次函数递归时都执行两次向量的修改。这种修改每次都把(vector-ref *V i*) ——*i* 的原值——向左移动一格，直到找到正确的位置为止。

图 43.3 描述了一个稍微更好一些的解决方案。在这里，第一行假设值 *a*、*b* 和 *c* 已经被正确地排序，也就是说，

```
(< a b ... c)
```

成立。另外，*d* 需要被插入，它的位置应该在 *a* 和 *b* 之间，也就是说，

```
(< a d b)
```

也成立。相应的解决方法是，比较 *d* 与 k+1 到 i 所有的元素，如果它们比 *d* 大的话，就把这些元素右移。最后，我们发现了 *a*（或者向量的左端），并且得到向量中的一个“空位”，而 *d* 就必须被插入到这个空位中（这个空位原来所包含的是 *b*）。图 43.3 中间的一行描述了这种情况。最后一行说明了 *d* 是怎样被放入 *a* 和 *b* 之间的。

开发一个函数 *insert*，按照这种描述的方法实现它所期望的效果。提示：新的函数必须读入 *d*，作为额外的参数。

习题 43.1.3 对于许多程序来说，我们可以交换 begin 表达式中子表达式的顺序，而程序还是可以正常运行。对 *sort-aux* 考虑这种想法：

```
;; sort2-aux : (vectorof number) N -> void
(define (sort2-aux V i)
  (cond
    [(zero? i) (void)]
    [else (begin
```

```
        (insert2 (sub1 i) V)
        (sort2-aux V (sub1 i)))]))
```

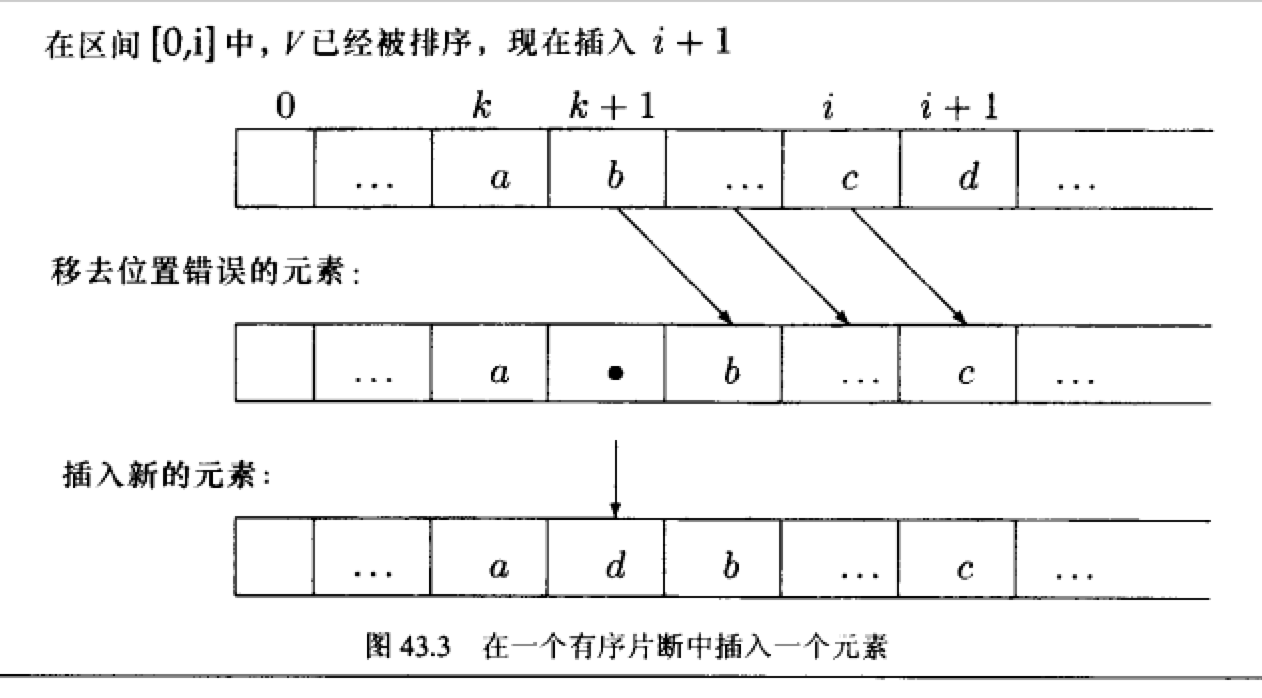


图 43.3　在一个有序片断中插入一个元素

这个顺序意味着 *sort2-aux* 先把元素(sub1 *i*)插入到 *V* 的某个已排序部分中，然后对 *V* 的其余部分排序。下面的图以图形的形式描述了这种情形：

	i-1	
…	*a*	…
左侧		右侧

这里描述的向量由三个部分组成：*a*（即字段(sub1 *i*)中的元素）左侧的片断以及右侧的片断。问题是，哪一个片断应该是已排好序的，而 *a* 又应该被插入到哪一个片断中。

考虑到 *sort2-aux* 递减它的第一个参数，从而从右向左扩展，这个问题的答案是，右侧的片断一开始是空的，从而被默认为升序排列的；左侧的片断还没有被排过序；所以 *a* 必须被插入到在右侧片断中它合适的位置中。

基于这样的观察，开发 *sort2-aux* 的一个效果说明。然后开发函数 *insert2*，使得 *sort2-aux* 能正确地对向量进行排序。

在第 25.2 节中，我们学习了 *qsort*，一个基于生成递归的函数。给定一个表，*qsort* 通过以下三个步骤构造出排序后的表：

1. 从表中选出一个元素，把它称为 *pivot*；

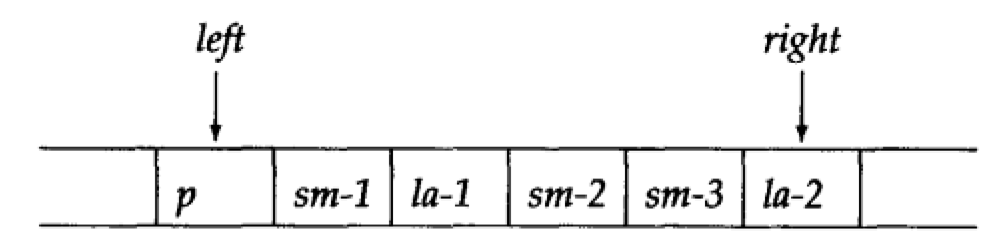
2. 建立两个子表：一个子表包含所有严格小于 *pivot* 的元素，另一个子表包含所有严格大于 *pivot* 的元素；

3. 使用同样的方式，对两个子表分别排序，然后连接两个子表，并把关键元素放在它们中间。

我们不难看出为什么返回值是有序的，为什么它包含了所有输入表中的元素，以及为什么这个过程能够停止。毕竟，在每一个阶段中，函数都从表中移去至少一个元素，使得两个子表都比原来的表短；最终这个表必然会变成空表。

图 43.4 描述了这种思想是怎样适用于处理向量在原来位置上排序的。在每一个阶段中，算法都处理向量的某个特定片断。算法选出第一个元素作为关键元素 *pivot*，重新安排这个片断，使得所有小于关键元素的元素都出现在 *pivot* 的左边，所有大于关键元素的元素都出现在 *pivot* 的右边。接下来，*qsort* 被调用两次：一次处理在 *left1* 和 *right1* 之间的片断，另一次处理在 *left2* 和 *right2* 之间的片断。因为这两个区间都比原来给定的区间要短，所以 *qsort* 最终会遇到空区间，并且停止。在 *qsort* 处理完所有的片断之后，排序就已经完成了；分割的过程已经把向量安排成了升序的片断。

一个向量片断，其中的关键元素是 p:



把这个向量片断分割成两个区域，区域之间由 p 隔开：

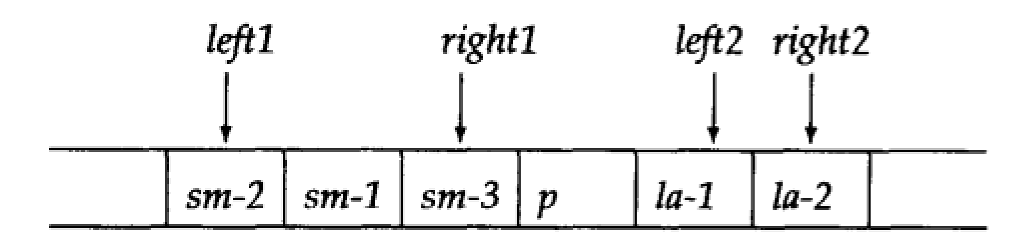


图 43.4 在原来位置快速排序的分割步骤

下面是 *qsort*，一个在原来的位置上对向量排序的算法定义：

```
;; qsort : (vectorof number) -> (vectorof number)
;; 效果：修改 V，使得它包含原来的元素
;; 但是按照上升的顺序排列
(define (qsort V)
  (qsort-aux V 0 (sub1 (vector-length V))))

;; qsort-aux : (vectorof number) N N -> (vectorof number)
;; 效果：对向量 V 中的区间[left, right]排序
;; 生成递归
(define (qsort-aux V left right)
  (cond
    [(>= left right) V]
    [else (local ((define new-pivot-position (partition V left right)))
            (begin (qsort-aux V left (sub1 new-pivot-position))
                   (qsort-aux V (add1 new-pivot-position) right)))]))
```

这里，主函数的输入是一个向量，所以它使用一个辅助函数来完成其工作。正如前面所提到的，这个辅助函数读入一个向量和两个边界。边界是向量的下标。一开始，两个边界分别是 0 和(sub1 (vector-length *V*))，这表示 *qsort-aux* 是用来处理整个向量的。

qsort-aux 的定义紧密地遵循算法的描述。如果 *left* 和 *right* 描述了一个长度为 1 或更短的片断，那么任务已经完成了。否则，函数就分割向量。因为分割步骤是一个单独的、复杂的过程，所以它需要一个单独的辅助函数。这个辅助函数必须既有效果，又有返回值，返回关键元素新的下标。给定这个下标，*qsort-aux* 就可以继续对 *V* 的区间[*left*，(sub1 *new-pivot-position*)]和[(add1 *new-pivot-position*)，*right*]排序。这两个区间都至少比原来的区间短一个元素，这就是 *qsort-aux* 的终止论证。

自然，现在关键的问题就是这个分割步骤，它由 *partition* 实现：

;; *partition :* (vectorof *number*) N N -> N

;; 求出关键元素的正确位置 *p*

;; 效果：重新安排向量 *V*，使得

;; ——*V* 中所有在[*left*，*p*)中的元素都比关键元素小

;; ——V 中所有在(p，right]中的元素都比关键元素大

(define (*partition V left right*) ...)

为了简单起见，我们选用给定的区间中最左的元素作关键元素。问题是，*partition* 怎样才能完成它的任务，例如，它是基于结构递归的还是基于生成递归的。另外，如果它是基于生成递归的，那么问题就是，生成的步骤完成了什么。

最好的方法是考虑一个例子，来看看分割步骤可以怎样来完成。第一个例子是只有六个数的短向量：

```
(vector 1.1 0.75 1.9 0.35 0.58 2.2)
```

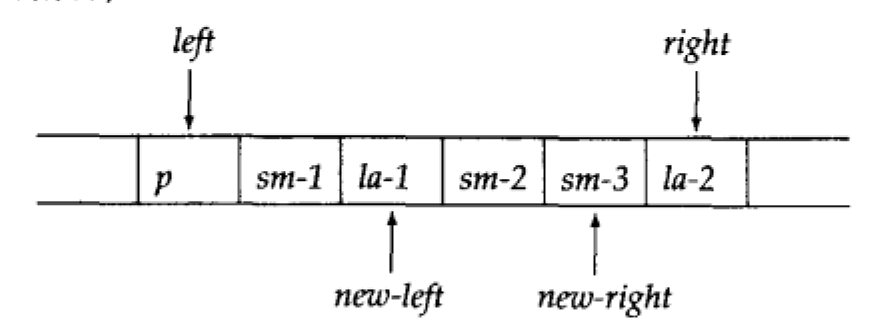
关键元素的位置是 0；关键元素是 1.1。边界是 0 和 5。有一个元素 1.9，显然不在正确的位置之上。如果把它和 0.58 交换，那么这个向量就几乎被分割好了：

```
(vector 1.1 0.75 0.58 0.35 1.9 2.2)
```

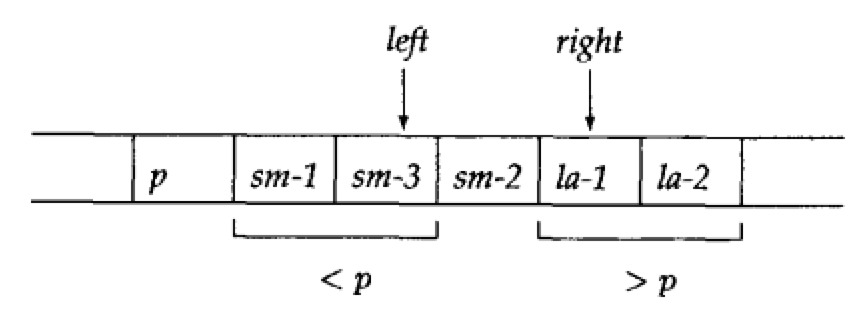
在这个修改后的向量中，惟一不在正确位置之上的元素就是关键元素自身。

图 43.5 描述了我们刚才所描述的交换过程。第一步，必须找出两个需要交换的值。要做到这一点，我们从 *left* 开始向右搜索第一个比关键元素大的元素。类似地，从 *right* 开始向左搜索第一个比关键元素小的元素。这两个搜索产生两个新的下标：*new-left* 以及 *new-right*。第二步，我们交换在 *new-left* 和 *new-right* 字段中的元素。结果，位于 *new-left* 的元素现在比关键元素小，而位于 *new-right* 的元素现在比关键元素大。最后一步，对新的、较小的区间继续交换过程。当在第一个步骤中产生的 *new-left* 和 *new-right* 的次序颠倒时（如同图 43.5 的最后一行所示的），那么我们就得到了一个基本被分割的向量（片断）。

求出分割 *partition* 的交换位置：



交换这两个元素，并对新的区间进行递归：



停止生成递归，最后进行整理：

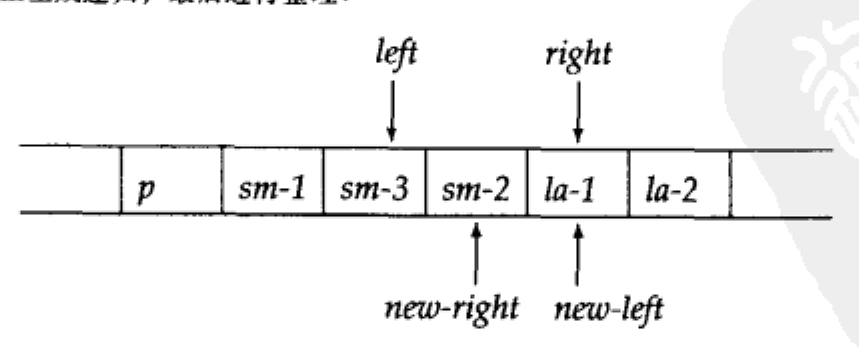


图 43.5　在原来位置快速排序的分割步骤

对这个例子的研究表明，*partition* 是一个算法，也就是说，它是一个基于生成递归的函数。遵照诀窍，我们必须提出并回答四个问题：

1. 平凡可解的问题是什么？

2. 相应的解答是什么？

3. 我们怎样从原来的一个问题生成新的、更容易解的问题？是一次生成一个问题还是多个？

4. 给定的问题的解是不是就是（某一个）新问题的解？或者，我们还需要执行一些额外的计算，在得出最后的解之前把这些解结合起来吗？另外，如果需要这样做，需要任何原问题的数据吗？

例子已经回答了问题 1、3 和 4。第一个步骤是求出下标 *new-left* 和 *new-right*。如果 *new-left* 小于 *new-right*，那么生成的工作就是交换这两个字段。接下来的过程就是对这两个新的边界进行递归。如果 *new-left* 大于 *new-right*，那么分割过程结束，只是关键元素的位置还没有调整。假设可以解决这个"平凡可解"的问题，那么我们知道整个问题就被解决了。

再来用一些例子研究问题 2。在第一个例子中，当向量变为

```
(vector 1.1 0.75 0.58 0.35 1.9 2.2)
```

时，我们就停止处理，此时的区间已被缩减为[2，4]。现在搜索 *new-left* 和 *new-right* 会得出 4 和 3。换句话说，

```
(<= new-right new-left)
```

成立。因为 *new-right* 指向了向量中比关键元素小的最右元素，所以交换字段 *new-right* 中的元素和原来最左端的边界，我们就把关键元素放到了正确的位置上：

```
(vector 0.35 0.75 0.58 1.1 1.9 2.2)
```

在接受这个看似简单的答案之前，我们再用一些额外的例子来检查一下它，特别是某些向量片断，其中的关键元素是向量中最大或最小的元素。下面就是一个这样的例子：

```
(vector 1.1 0.1 0.5 0.4)
```

假设初始的区间是[0，3]，关键元素是 1.1。那么，向量中所有其他的元素都比关键元素小，这意味着最后它应该位于最右侧的位置上。

我们的处理过程显然会得出 *new-right* 是 3。毕竟，0.4 比关键元素要小。不过，对 *new-left* 的搜索就不一样了。既然向量中没有比关键元素大的元素，最终生成的下标就是 3，也就是这个向量中最大的合法下标。这时搜索就停止了。幸运的是，这时的 *new-left* 和 *new-right* 是相等的，这意味着分割过程可以停止，还意味着我们可以交换关键元素与 new-right 中的元素。如果我们这样做，就可以得到一个正确分割的向量：

```
(vector 0.4 0.1 0.5 0.4 1.1)
```

在第三个例子向量中，所有的元素都比关键元素大：

```
(vector 1.1 1.2 3.3 2.4)
```

在这种情况下，对 *new-left* 和 *new-right* 的搜索会发现关键元素已经在正确的位置上了。而且事实也确是如此。对 *new-left* 的搜索在字段 1 停止，这就是第一个包含大于关键元素的元素的字段。对 *new-right* 的搜索停止于 0，因为它是最小的合法下标，而且搜索必须在此停止。结果，*new-right* 又一次指出了在被分割的向量（的片断）中要包含关键元素的字段。

简而言之，例子说明了这几件事：

1. `Partition` 的终止条件是 `(<= new-right new-left)`。

2. *new-right* 的值就是关键元素最后的位置，而原来关键元素的位置是区间中最左的位置。只需交换这两个字段的内容就可以解决问题。

3. 对 *new-right* 的搜索从最右侧的边界开始，向左直到找到一个比关键元素小的元素，或者知道它到达了最左侧的边界。

4. 对应地，对 *new-left* 的搜索从最左侧的边界开始，向右直到找到一个比关键元素大的元素，或者知道它到达了最右侧的边界。

另外，这两个搜索是复杂任务，需要它们自己的函数。

我们现在可以逐步把讨论翻译成 Scheme。第一步，分割过程是一个函数，它不仅需要向量以及某个区间，还需要向量原来的最左位置以及它的内容。这表明我们需要使用 **local** 局部定义的函数和

变量：

```
(define (partition V left right)
  (local ((define pivot-position left)
          (define the-pivot (vector-ref V left))
          (define (partition-aux left right)
        ...))
    (partition-aux left right)))
```

另一种可选的方法是使用辅助函数，这个辅助函数除了需要读入向量和当前的区间之外，还需要读入关键元素原来的位置。

第二步，这个辅助函数读入一个区间的边界。它立即由这对边界生成一对新的下标：*new-left* 和 *new-right*。如前所述，对两个新的边界的搜索是复杂任务，需要它们自己的函数：

```
;; find-new-right : (vectorof number) number N N [>= left] -> N
;; 求出一个在 left （包含）和 right 之间的下标 i，
;; 使得(< (vector-ref V i) the-pivot)成立
(define (find-new-right V the-pivot left right) ...)

;; find-new-left : (vectorof number) number N N [<= right] -> N
;; 求出一个在 left 和 right （包含）之间的下标 i，
;; 使得(> (vector-ref V i) the-pivot)成立
(define (find-new-left V the-pivot left right) ...)
```

使用这两个函数，*partition-aux* 可以生成新的边界：

```
(define (partition V left right)
  (local ((define pivot-position left)
          (define the-pivot (vector-ref V left))
          (define (partition-aux left right)
            (local ((define new-right (find-new-right V the-pivot left right))
                    (define new-left (find-new-left V the-pivot left right)))
          ... )))
    (partition-aux left right)))
```

接下来，剩余的定义只需简单地把我们的讨论翻译成 Scheme。

图 43.6 给出了 *partition*、*partition-aux* 以及 *find-new-right* 的完整定义；函数 *swap* 的定义位于图 43.2 中。搜索函数的定义使用了一种特殊的结构递归，基于自然数的一个子集的结构递归，其中的自然数被限制在函数的参数之间。因为搜索函数是基于一种不常用的设计诀窍的，所以我们最好单独地对它们进行设计。不过，它们只是在 *partition* 的环境中应用，这意味着在完成了它们的设计之后，我们还要把它们的定义整合到这个定义之中。

习题

习题 43.1.4　完成 *find-new-left* 的定义。这两个定义有着相同的结构；开发它们共同的抽象。

使用 *find-new-right* 和 *find-new-left* 的定义，给出 *partition-aux* 的一种终止论证。

使用例子，开发 *partition* 的测试。回忆一下，这个函数计算关键元素正确的位置，并且重新安排一个向量的片断。用布尔值表达式表示测试。

在完成了函数的测试之后，把 find-new-right 和 find-new-left 整合到 partition 之中，并消除多余的参数。

最后，测试 *qsort*，并给出一个单独的函数定义，实现在原来位置上的快速排序算法。

习题 43.1.5 开发函数 *vector-reverse!*，该函数把某个向量倒转过来；它的返回值是倒转后的向量。

提示： 从向量的两端开始交换元素，直到没有元素可以交换为止。

习题 43.1.6 经济学家、气象学家等经常会测量各种东西，得到时间序列，需要计算“n 个元素的平均”或者“滤波”等。假设我们有某种货物一周的价格表：

```
;; partition : (vectorof number) N N -> N
;; 求出关键元素的正确位置 p
;; 效果：重新安排向量 V，使得
;; ——V 中所有在[left，p)中的元素都比关键元素小
;; ——V 中所有在(p，right]中的元素都比关键元素大
;; 生成递归
(define (partition V left right)
  (local ((define pivot-position left)
          (define the-pivot (vector-ref V left))
          (define (partition-aux left right)
            (local ((define new-right (find-new-right V the-pivot left right))
                    (define new-left (find-new-left V the-pivot left right)))
              (cond
                [(>= new-left new-right)
                 (begin
                   (swap V pivot-position new-right)
                   new-right)]
                [else ; (< new-left new-right)
                 (begin
                   (swap V new-left new-right)
                   (partition-aux new-left new-right))]))))
    (partition-aux left right)))

;; find-new-right : (vectorof number) number N N [>= left] -> N
;; 求出一个在 left 和 right （包含）之间的下标 i，
;; 使得(< (vector-ref V i) the-pivot)成立
;; 结构递归：参见课文
(define (find-new-right V the-pivot left right)
  (cond
    [(= right left) right]
    [else (cond
            [(< (vector-ref V right) the-pivot) right]
            [else (find-new-right V the-pivot left (sub1 right))])]))
```

图 43.6 重新排列一个向量的片断，使之被分割为两部分

1.10　　1.12　　1.08　　1.09　　1.11

计算相邻三个元素的平均，结果如下：

1.10	1.12	1.08	1.09	1.11
	1.10	$\frac{3.29}{3.00}$	$\frac{3.28}{3.00}$	

表的末端是没有平均值的，这意味着 k 个元素的序列有 k－2 个平均值。

开发函数 *list-3-average*，该函数计算某个数表中（连续）三个元素的（滑动）平均值。更确切地说，我们用一个表来表示某货物的价格序列，即函数 *list-3-averages* 读入表

```
(list 1.10 1.12 1.08 1.09 1.11)
```

返回

```
(list 1.10 329/300 82/75)。
```

设计函数 *vector-3-averages*，该函数计算向量中（连续）三个元素的（滑动）平均值。既然向量是可变的，我们既可以返回一个新的向量，也可以修改现有的向量。

开发这个函数的两个版本：一个版本返回一个新的向量，另一个版本修改它所处理的向量。

警告：这是一道很难的习题，请对三个版本的函数和设计它们的复杂性进行比较。

习题 43.1.7　这一节中所有的例子处理的都是向量片断，也就是自然数的区间。处理区间需要有区间的起点和终点，此外，如 *find-new-right* 和 *find-new-left* 的定义所示，还需要有遍历的方向。另外，处理意味着把某个函数作用于区间中的每一个位置。

下面是一个处理区间的函数：

```
;; for-interval : N (N -> N) (N -> N) (N -> X) -> X
;; 对 i, (step i), ……, 计算(action i (vector-ref V i))
;; 直到(end? i)成立为止（包含）
;; 生成递归: step 生成新的值, end?探测终止
;; 并不能保证终止
(define (for-interval i end? step action)
  (cond
    [(end? i) (action i)]
    [else (begin
        (action i)
        (for-interval (step i) end? step action))]))
```

这个函数读入起点下标 *I*、一个判断是否到达了区间的终点的函数、一个生成下一个下标的函数，以及一个作用于中间的每一个位置的函数。假设(*end?* (*step* (*step* ... (*step* *i*) ...)))成立，那么 *for-interval* 满足如下的等式：

```
  (for-interval i end? step action)
= (begin (action i)
         (action (step i))
        ...
        (action (step (step ... (step i) ...))))
```

比较这个函数定义与 map 的定义。

使用 *for-interval*，无需使用辅助函数就可以开发出（某些）处理向量的函数。使用 *for-interval* 的方法就与使用 map 处理某个表中所有的元素一样。下面是一个函数，它把向量的每个字段加 1：

```
;; increment-vec-rl : (vector number) -> void
;; 效果: 把 V 中的每一个元素增加 1
(define (increment-vec-rl V)
  (for-interval (sub1 (vector-length V)) zero? sub1
                (lambda (i)
                  (vector-set! V i (+ (vector-ref V i) 1)))))
```

这个函数处理区间[0，(sub1 (vector-length *V*))]。这里左边界由终止测试 zero?判定，而起点是(sub1

(vector-length *V*))，也就是向量最右侧的合法下标。*for-interval* 的第三个参数是 sub1，它确定了遍历的方向是从右向左，直至下标为 0 为止。最后，函数的行为是修改第 *i* 个字段的内容，把它加 1。

下面是另一个函数，它对向量有着同样的可见效果，但是按照不同的顺序处理：

```
;; increment-vec-lr : (vector number) -> void
;; 效果：把 V 中的每一个元素增加 1
(define (increment-vec-lr V)
  (for-interval 0 (lambda (i) (= (sub1 (vector-length V)) i)) add1
              (lambda (i)
                (vector-set! V i (+ (vector-ref V i) 1)))))
```

这个函数的起点是 0，终点是 *V* 最右的合法下标。add1 函数决定了向量的处理是从左向右进行的。

使用 *for-interval*，开发下列函数：

1. *rotate-left*，该函数把向量中所有的元素都移动到左侧相邻的字段中，而第一个元素被移到最后一个字段中；

2. *insert-i-j*，该函数把所有在下标 *i* 和 *j* 之间的元素都移动到右侧相邻的字段中，而最右端的元素被插入到第 *i* 个字段中（比较图 43.3）；

3. *vector-reverse!*，该函数交换一个向量的左半部分和右半部分；

4. *find-new-right*，也就是图 43.6 的另一种定义；

5. *vector-sum!*，该函数使用 set!计算某个向量中数的总和（提示：参见第 37.3 节）。

最后的两个任务表明，*for-interval* 对于没有可见效果的计算来说是很有用的。当然，习题 29.4 表明没有必要定义一个像 *vector-sum!*这样笨拙的函数。

这些函数中的哪些可以使用习题 41.2.17 中的 *vec-for-all* 定义？

循环结构：许多程序设计语言（必须）提供类似 *for-interval* 的函数作为内建的结构，并迫使程序员使用它们来处理向量。结果，许多程序被迫使用不必要的 set!，从而需要进行复杂的时间推理。

43.2 带循环的结构体集合

在我们的世界中，由许多事物都是以循环的形式与其他事物相关联的。我们都有父母；我们的父母都有孩子。一台计算机可能会与另一台计算机相连接，而那台计算机又会连接回原来的计算机。我们也已经看到过引用其他数据定义的数据定义。

既然数据表达真实世界中事物的信息，那么我们就会遇到这样的情形，需要设计一种包含循环关系的结构体类型。在过去，我们回避了这个问题，或者使用一个小诀窍来表示集合。这个小诀窍就是使用间接性。例如，在第 28.1 节中，我们把每个结构体都和一个数相结合，然后建立一个符号和对应结构体的表格，并把符号放入结构体。接下来，当需要查询某个结构体是否连接到另一个结构体时，我们就提取出相关的符号，在表格中查找该符号的结构体。这种间接的使用虽然允许我们用循环的关系来表示相互引用的结构体，或者表示有循环关系的结构体，但是它也导致了笨拙的数据表示法和笨拙的程序。这一节示范了我们可以用结构体的变化器来简化集合的表示。

要使这种思想变得具体，我们来讨论两个例子：家谱树和简单图。考虑家谱树的例子。迄今为止，我们已经使用过两种类型的家谱树来记录家庭关系了。第一种是祖先树，它把人和他的父母、祖父母等相关联。第二种是后代树，它把人和他的孩子、孙子等相关联。换句话说，我们避免了把这两种家谱树结合成一棵树，而在真实的世界中就只有一种家谱树。回避这种联合表示的原因是显然的。翻译成我们的数据语言，一棵联合树需要有一种结构体，在这种结构体中，父亲应当包含他孩子的结构体，而每一个孩子的结构体都应该包含这个父亲的结构体。在过去，我们无法建立这样一种结构的集合。有了结构体的变化器，我们现在可以建立这样的东西了。

能够使这个讨论变得具体的结构体定义是：

```
(define-struct person (name social father mother children))
```

我们的目标是建立由 *person* 结构体构成的家谱树。*person*（人）结构体有五个字段。每个字段的内容都由如下的数据定义指定：

> family-tree-node（家谱树结点，简称 ftn）是下列两者之一：
>
> 1. false。
> 2. 一个 person。
>
> person 是结构体：
>
> (make-person n s f m c)
>
> 其中 n 是符号，s 是数，f 和 m 是 ftn，c 是(listof person)。

与以往一样，*family tree node* 定义中的 false 表示丢失了家谱树中某个部分的信息。

只使用 *make-person*，我们无法在家谱树中某个父亲和他的孩子之间建立相互的引用。假设使用祖先树的方式，也就是先建立父亲节点，那么我们就无法在 *children* 字段中填写任何的孩子，因为按照假设，相应的结构体还不存在。反过来，如果使用后代树的方式，先为某个父亲所有的孩子建立结构体，那么这些结构体就无法包含任何有关父亲的信息。

这表明，对于这种类型的数据，简单的构造器并不足以定义出它们。作为代替，我们应当定义一个一般化的构造器，它不仅建立 *person* 结构体，在条件允许的情况下，还要适当地对结构体进行初始化。开发这个函数，最好按照真实的世界进行，也就是随着某个孩子的诞生，在家谱树中建立一个新的条目，然后记录这个孩子的父母，再在现有的父母条目中记录他们有了一个新的孩子。下面就是这样一个函数的说明：

```
;; add-child! : symbol number person person -> person
;; 为一个新诞生的孩子建立 person 结构体
;; 效果：把新的结构体添加到这个孩子的父亲和母亲中
(define (add-child! name soc-sec father mother) ...)
```

这个函数的任务是为一个新诞生的孩子建立新的结构体，并把这个结构体添加到现有的家谱树中。该函数读入一个孩子的名字、社会保障号码以及代表他的父亲和母亲的结构体。

设计 *add-child!*的第一个步骤是为这个孩子建立新的结构体：

```
(define (add-child! name soc-sec father mother)
  (local ((define the-child
            (make-person name soc-sec father mother empty)))
    ...))
```

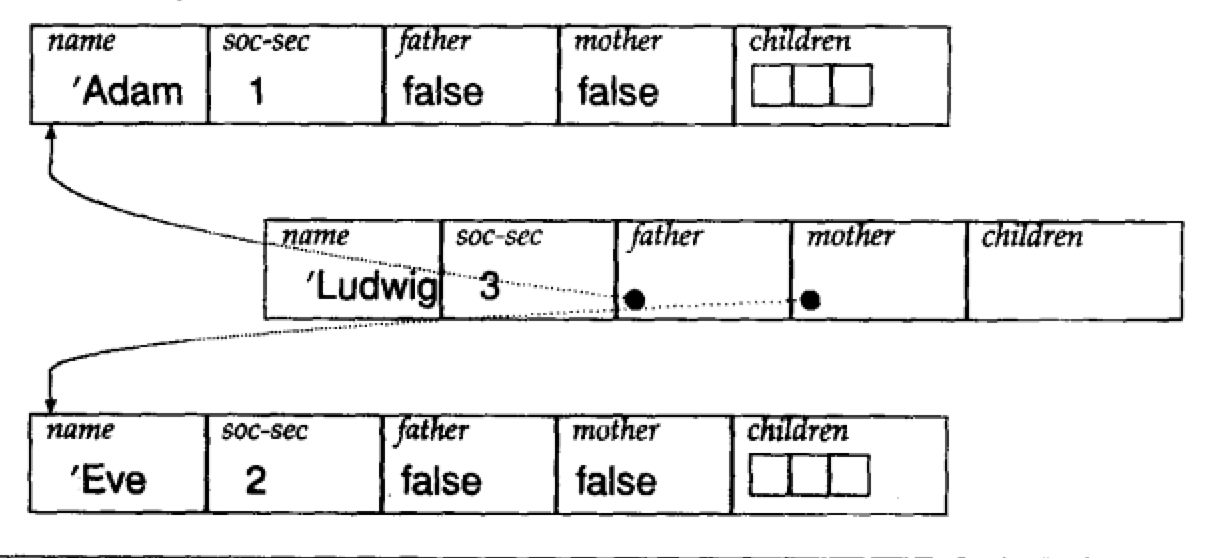
这覆盖了合约的第一个部分。通过在 local 表达式中对结构体命名，我们就可以在表达式的主体中修改它了。

设计 *add-child!*的第二个步骤是写出 local 表达式的主体，执行所需的效果：

```
(define (add-child! name soc-sec father mother)
  (local ((define the-child
            (make-person name soc-sec father mother empty)))
    (begin
      (set-person-children! father
                            (cons the-child (person-children father)))
      (set-person-children! mother
                            (cons the-child (person-children mother)))
      the-child)))
```

既然这个函数有两个效果，而且用途说明还指定了一个返回值，那么 local 表达式的主体就应该是一个 begin 表达式，其中包含三个子表达式，第一个子表达式修改 *father*，把 *the-child* 添加到它的孩子表中，第二个子表达式对 *mother* 作类似的修改，最后一个子表达式生成所需的返回值。





… 以及修改 'Adam 和 'Eve 结构体之后的树：

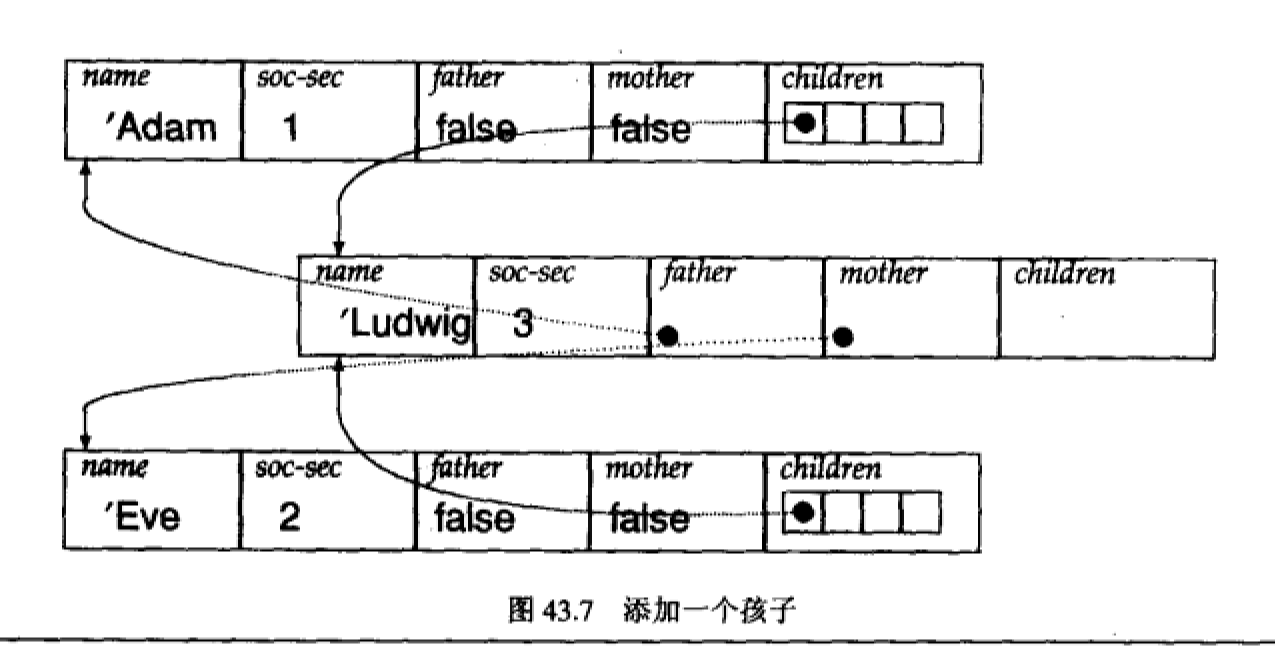


图 43.7 添加一个孩子

图 43.7 描述了 *add-child!* 调用的计算过程：

```
(add-child! 'Ludwig 3
            (make-person 'Adam ... ... ...)
            (make-person 'Eve ... ... ...))
```

图的上半部分显示了新的'Ludwig 结构体，以及它是怎样引用 *father* 和 *mother* 结构体的。如同第 14.1 节，该图使用箭头表示家谱树中节点的相互关联。但是现在，选用箭头不仅仅是为了方便，而是必需的。正如图的下半部分所示的，*add-child!* 中的结构体变化器修改了 *father* 和 *mother* 的 *children* 字段，在这个字段的表中添加了一个元素，也就是'Ludwig 的结构体。如果不使用箭头，就无法绘制出这种结构体的布局，因为我们不可能绘制出两个相互嵌套的结构体。

有了 *add-child!*，每次添加一个孩子，就可以建立家谱树。我们需要从中学习的是，如何设计处理这种新家谱树类型的函数。在这个例子中，我们总是可以选用我们以前所使用的两种观点中的一种：祖先家谱树或者后代家谱树。每种观点都忽略结构体中特定的字段。一旦选定了一种观点，就可以按照已知的诀窍设计所需的函数。即使我们需要在新的家谱树中使用双向关系，设计函数也很简单，只需按照真实世界中的家庭关

系写出辅助函数，然后相应地结合这些辅助函数。接下来的一些练习题就是这个原则的示范。

习题

习题 43.2.1 修改 *add-child!*，使它具有如下的合约：

```
;; add-child! : symbol number ftn ftn -> person
```

在其他方面，这个函数的行为与原来的函数相同。

一旦得到了修改后的函数，就不再需要 *make-person* 了。我们可以直接使用 *add-child!*来建立各种形式的 *person* 结构体。

把图 14.1 中的家谱树转化成新的表示法；只使用新修改后的 *add-child!*函数。

习题 43.2.2 开发函数 *how-many-ancestors*，该函数读入一个家谱树节点，判断其有多少个祖先。这个节点本身也算作是一个祖先。

习题 43.2.3 开发 *how-many-descendants*，该函数读入一个家谱树节点，判断其有多少个后代。这个节点本身也算作是一个后代。

习题 43.2.4 开发 *names-of-cousins*。这个函数读入一个 *person*，返回其堂、表兄弟姐妹的名字。

提示：（1）不要忘记使用 Scheme 内建的函数来处理表。（2）使用足够大的家谱树来测试这个函数。（3）在测试步骤中，比较辅助函数返回的名字和它的期望值。因为结构体是互相引用的，所以我们很难自动地对它们进行比较。作为选择，使用 eq?——Scheme 的内涵相等谓词——来比较结构体。为什么这是可行的？

在第 28.1 节和第 30.2 节中，我们遇到了图的表示和遍历问题。回忆一下，图是节点以及节点之间的连接的集合。图的遍历问题是要判断图中有没有一条从标号为 *orig* 的节点到标号为 *dest* 的节点的路线。在简单图中，每个节点都正好有一条连接，从该节点到另一个节点。

起初，我们使用节点（有名字的节点）的表来表示图。如果某个节点有一条到达其他节点的连接，那么相应的结构体（第一个节点的结构体）就包含第二个节点的名字，而不是包含第二个节点自身。习题 30.2.3 引入了基于向量的（图的）表示法。这个表示法依然使用了间接小诀窍，所以如果我们想从某个节点移动到另一个节点，必须先在表格中查找连接。

使用结构体变化器，我们可以除去这种间接性，建立互相包含的节点结构体，即使图中包含了循环。要具体理解这是怎样工作的，先来看两个问题：建立如图 30.3 中简单图的模型，以及如何设计在这种图中查找路线的程序。首先，我们需要节点（node）的结构体定义：

```
(define-struct node (name to))
```

name 字段记录节点的名字，*to* 字段指定这个节点连接到哪一个节点。还需要一个数据定义：

> 简单图（节点）（simple-graph-node）是结构体：
>
> (make-node n t)
>
> 其中 n 是符号，t 是 node。

这个数据定义的奇特之处是，它是自我引用的，但是它并不是由多个子句构成的。这马上就提出问题：我们怎样来建立一个符合该定义的节点。显然，调用 make-node 并不可行；作为代替，我们需要定义一个一般化的构造器，直接设置节点的 *to* 字段。

这个一般化的构造器读入一个 *node* 的原子数据，并由此构造一个合法的结构体：

```
;; create-node : symbol -> node
;; 建立一个合法的简单图节点，其 name 字段中包含 a-name
(define (create-node a-name)
```

```
(local ((define the-node (make-node a-name false))) ...))
```

我们自然会想到，把这个节点自身放入 *to* 字段中。换句话说，这个一般化的构造器建立一个包含自身的节点：

```
;; create-node : symbol -> node
;; 建立一个简单图节点，包含 a-name 和它自身
(define (create-node a-name)
  (local ((define the-node (make-node a-name false)))
    (begin
      (set-node-to! the-node the-node)
      the-node)))
```

这个一般化的构造器使用普通的构造器来建立节点，正确地初始化 *name* 字段，并把 false 放入 *to* 字段。虽然按照我们的数据定义，这样做是不正确的，但是这是可以接受的，因为在 local 表达式的主体中，这一点马上会被改正。因此，调用 *create-node* 会返回预期的 *node*。

有了 *create-node*，我们可以建立图中的节点，但是还不能建立它们之间的关联。要连接两个节点，我们必须修改其中一个结构体的 *to* 字段，使它包含另一个节点。虽然这样做基本上达到了目的，但问题是我们如何鉴别节点。家谱树的例子给出了一种解决方案，也就是为每一个节点引入一个变量。另一种方法就是原来操作简单图的方法，用节点（或者是连接的符号对）的表或者节点的向量来表示图。这里，我们选用第二种方法：

simple-graph（简单图）是(listof node)。

假设我们有了所有节点的表，比方说 *the-graph*，以及查找具有给定名字的节点的函数，比方说 *lookup-node*，就可以使用结构体的修改来建立（从某个结构体到某个结构体的）连接：

```
(set-node-to! (lookup-node from-name the-graph)
              (lookup-node to-name the-graph))
```

使用一个辅助函数，就可以更加简单地建立两个节点之间的连接：

```
;; connect-nodes : symbol symbol graph -> void
;; 效果：修改 name 字段为 from-name 的结构体中的 to 字段，
;; 使得它包含 name 字段为 to-name 的结构体
(define (connect-nodes from-name to-name a-graph)
  (set-node-to! (lookup-node from-name a-graph)
                (lookup-node to-name a-graph)))
```

定义 *lookup-node* 是一道结构函数设计的练习题，不过最好的定义方法是使用 Scheme 的 assf 函数，该函数就是对这种情形的抽象。

现在可以把简单图转换成 Scheme 表示了。假设从图 30.3 中的图开始，这里我们给出该图的表格形式：

从	A	B	C	D	E	F
到	B	C	E	E	B	F

第一个步骤是建立所有节点的表，并给它命名。第二个步骤是按照这个表格建立连接。图 43.8 给出了相应的 Scheme 表达式。只需直接翻译图的表格形式就可以得出这个表达式。没有必要重新连接'F 节点，因为它已经被连接到其自身了。

```
;; create-node : symbol -> node
;; 建立一个包含自身的简单图节点
(define (create-node name)
  (local ((define the-node (make-node name false)))
    (begin
      (set-node-to! the-node the-node)
      the-node)))

;; connect-nodes : symbol symbol graph -> void
;; 效果：修改名为 from-name 的结构体中的 to 字段，
;; 使它包含名为 to-name 的结构体
(define (connect-nodes from-name to-name a-graph)
  (set-node-to! (lookup-node from-name a-graph)
                (lookup-node to-name a-graph)))

;; lookup-node : symbol graph -> node or false
;; 查找 a-graph 中名为 x 的节点
(define (lookup-node x a-graph)
  ...)

;; the-graph : graph
;; 所有可用节点的表
(define the-graph
  (list(create-node 'A)
       (create-node 'B)
       (create-node 'C)
       (create-node 'D)
       (create-node 'E)
       (create-node 'F)))

;; 建立图：
(begin
  (connect-nodes 'A 'B the-graph)
  (connect-nodes 'B 'C the-graph)
  (connect-nodes 'C 'E the-graph)
  (connect-nodes 'D 'E the-graph)
  (connect-nodes 'E 'B the-graph))
```

图 43.8　通过修改建立一个简单图

习题

习题 43.2.5　分别使用第二部分中“方框套方框”以及第三部分中“方框和箭头”的方法，画出(*create-node* **'A**)的图形。

习题 43.2.6　不建立所有节点的表，而直接把给定的简单图转换成 Scheme 表示法。

习题 43.2.7　开发函数 *symbolic-graph-to-structures*。这个函数读入一个对（连接）的表，建立一个 *graph*。

例子：

```
(define the-graph
  (symbolic-graph-to-structures '((A B) (B C) (C E) (D E) (E B) (F F))))
```

计算这个定义等价与计算图 43.8 中的定义。

一旦得到了简单图的表示法，我们就可以把注意力转到下一个问题上，也就是寻找给定的图中从一

个节点到另一个节点的路线。回忆第 30.2 节中这个问题原来的说明：

```
;; route-exists? : node node simple-graph -> boolean
;; 判断在 sg 中有没有一条从 orig 到 dest 的路线
(define (route-exists? orig dest sg) ...)
```

当然，在新的环境中我们必须重新解释数据类型的名字，但是在其他的方面，这个说明都很好。

原来的函数开发阐明了两种新的思想。第一种新思想，这个函数使用了生成递归。一旦我们知道了 *orig* 和 *dest* 是不同的节点，搜索就会从 *orig* 所连接到的节点重新开始。第二种新思想，这个函数需要用累积器来记住哪些节点已经被访问过了。如果没有累积器，函数可能会反复地访问同样的节点。

所以，我们从生成递归的模板开始（定义新的函数）：

```
(define (route-exists? orig dest sg)
  (cond
    [(eq-node? orig dest) true]
    [else
     (route-exists? ... orig 所连接的节点... dest sg)]))
```

函数 *eq-node?*判断两个节点是否相等；这里，我们也可以使用 eq?——Scheme 的内涵相等谓词，或者假设节点的名字是惟一的，我们可以比较两个节点的名字。如果两个节点是同一个，那么路线就存在。如果不相等，我们可以移动到 *orig* 所连接的节点，从而生成一个新问题，它可能对解决问题有所帮助。在第 30.2 节中的图的表示法下，这需要在 *sg* 中进行查找。在新的图的表示法下，连接是 *node* 表示法的一个部分。因此我们可以使用 *node-to*，而无须在 *sg* 中查找：

```
(define (route-exists? orig dest sg)
  (cond
    [(eq-node? orig dest) true]
    [else (route-exists? (node-to orig) dest sg)]))
```

这个函数定义说明，到目前为止，*sg* 毫无用处。因为在新的图的表示法下，节点包含了它的邻居，而邻居又包含了它的邻居，以此类推，所以没有必要使用表格。

与第 30.2 节中原来的函数一样，这个函数的终止论证不能成立。要理解为什么我们的新函数可能无法终止，请观察它的定义。定义中并没有包含 false，而且函数没有可能返回 false——即使我们知道它应该返回 false。例如，我们知道这张简单图中并不包含从'F 到'A 的路线，如果观察

```
(route-exists? (lookup-node the-graph 'F)
               (lookup-node the-graph 'A))
```

的计算中发生了什么，我们会发现 *route-exists?*反复地访问节点'F。简而言之，它忘记了它已经处理过的节点。

我们知道给 *route-exists?*添加一个累积器就可以解决这个知识丢失的问题，但是这需要另一种形式的表格查找。使用一个结构体变化器来记录 *route-exists?*函数的访问，我们就可以做得更好。要做到这一点，*node* 结构体需要一个额外的字段；我们称这个字段为 *visited*（已访问过）：

```
(define-struct node (name visited to))
```

一开始，这个字段中包含了 false。随着 *route-exists?*访问某个节点，它会在这个字段中放入 true：

```
(define (route-exists? orig dest sg)
  (cond
    [(eq-node? orig dest) true]
    [(node-visited orig) false]
    [else
```

```
(begin
  (set-node-visited! orig true)
  (route-exists? (node-to orig) dest sg))]))
```

要使用这个新的知识，函数检查结构体的这个新字段，以此作为新的终止条件。如果以前已经访问过 *orig* 了，那么路线就不存在，因为函数已经在图中找到了一个循环。

这个例子中的第二个结构体修改举例说明了两种思想。第一种思想，结构体的修改可以取代基于表格的累积器。不过一般而言，我们最好学习基于表格的累积器，在基于对累积的知识充分掌握的基础上再添加结构体修改。第二种思想，结构体的修改可以在生成递归的终止测试中起作用。毕竟，状态的改变是由记住函数调用中的事情而引起的，而终止测试必须发现事情有没有被改变。虽然这种结合并不常见，但是它很有用，而且在学习算法时它还会反复出现。

习题

习题 43.2.8 函数 *route-exists?*假设所有节点的 *visited* 字段初始值都是 false。可是，对该函数的一次调用就会把图中的某些字段设为 true。这意味着该函数不能连续被调用两次。

开发 *route-exists?*的一个修改版本，该版本的函数说明与原来的函数相同，但是它在开始搜索路线之前把所有的 *visited* 字段都设置为 false。

假设图中共有 N 个节点，求出这个新函数的抽象运行时间。

习题 43.2.9 开发函数 *reachable*，该函数读入简单图中的一个节点。它的效果是把所有从这个给定节点出发可以访问到的节点的 *visited* 字段设置为 true，并保证所有其他节点的 *visited* 字段为 false。

习题 43.2.10 开发 *make-simple-graph*，管理一个 local 局部定义的图的状态的函数。这个函数读入一张简单图，其数据形式为符号对的表：(listof (list *symbol symbol*))。该函数支持四种服务：

1. （使用节点的名字）添加节点，该节点连接到某个已存在的节点；
2. （使用节点的名字）修改某个节点的连接；
3. 判断两个节点之间是否存在一条路线；
4. 删除从某个给定节点出发所不能到达的（所有）节点。

提示： 该管理器不应使用表，而应使用节点的序列，类似于第 41.3 节中的 *hand* 结构体。节点的序列基于如下的结构体：

```
(define-struct sequence (node next))
```

序列类似于表，但是它支持结构体的修改。

这一节中的讨论证实了设计诀窍的有效性，即使是针对相互引用的结构体集合，设计诀窍也是有用的。这里最重要的经验是，这种情况会调用一个一般化的构造器，即一个建立结构体、并且马上建立所需的连接的函数。一般化的构造器是第 35 章中初始化函数的对应物；我们在第 41.3 节中也见到过这种思想，那一节建立单张牌的 hand。在许多情况下，例如简单图等，我们可能还需要引入辅助函数，用来再一次修改结构体。一旦有了这些函数，就可以使用标准的诀窍，包括引入额外结构体字段的诀窍。

43.3 状态的回溯

第 28 章介绍了回溯算法。回溯算法是一种递归函数，它在递归的过程中生成新的问题，而不是使用其输入数据的某个部分。有时候，算法可能需要在解决问题的方法的多个分支中进行选择。某些选择可能不会有什么结果。在这种情况下，算法需要回溯。更确切地说，它可以重新开始，搜索另一个分支，检查它能不能取得成功。

如果某个问题的数据表示使用结构体或者向量，回溯算法就可以使用结构体的修改来测试解的不同

分支。这里的关键是要设计一对函数，一个函数修改问题表示法的状态，另一个函数在尝试失败时取消这种修改。在这一节中，我们讨论两个这种类型的例子：皇后问题和单人棋子游戏。

回忆第 28.2 节中的皇后问题。这个问题的目标是把 *n* 个皇后放到某个 m×m 的棋盘上，使得皇后之间相互不构成威胁。在国际象棋中，皇后会威胁它所在的行、列和两条斜线上的所有位置。图 28.4 描述了 8×8 的棋盘上的一个皇后的概念。

在第 28.2 节中，我们用表来表示棋盘。后来在学习向量时，我们还在习题 29.3.14 中开发了如下的基于向量的表示法：

```
;; chess-board（棋盘）CB 是(vectorof (vectorof boolean))
;; 其中所有向量的长度都相同。
;; make-chess-board : N -> CB
(define (make-chess-board m)
  (build-vector m (lambda (i) (build-vector m (lambda (j) true)))))
```

初始值 true 表示现在把皇后放到相应的字段中是合法的。放置皇后的算法把一个皇后放到给定的棋盘的某个可用字段上，然后建立一个新的棋盘，表示添加了一个皇后。这个过程会被重复，直到所有的皇后都被放置好，在这种情况下问题就被解决了。或者皇后还没有放完，但是已经没有位置可以用来放皇后了。在这种情况下，算法移去最后一个被添加的皇后，为它选择其他可用的字段。如果没有其他可用的字段，算法就进一步回溯。如果算法不能再回溯了，它就产生一个错误消息。

一方面，在每一个阶段都建立一个新的棋盘是可行的，因为以后可能会发现这个棋盘是错误的，在这种情况下，我们又需要从原来的棋盘重新开始（递归）。另一方面，对一个人来说，他更希望做的是把皇后放到棋盘上，然后，如果这个位置被证实是错误的，他会把皇后移走。这样，皇后问题的例子就表明，计算机程序有能力建立许多可选的“世界”，而人类在这方面的能力非常有限 [1]，所以人类的想象力是受到限制的。不过，我们还是有必要研究一下，在我们的符号集中加上向量的修改后，我们可以怎样更好地模仿人类处理皇后问题。

习题

习题 43.3.1 把另一个皇后放到棋盘上意味着棋盘上的某些字段必须被设置为 false，因为它们所对应的位置会受到给定皇后的威胁，从而不能再放入皇后了。放置皇后是一个函数，该函数读入一个棋盘以及新皇后的下标：

```
;; place-queen : CB N N -> void
;; 效果：把 CB 中被位于 i 行、j 列的皇后所威胁
;; 的字段设置为 false
(define (place-queen CB i j) ...))
```

提示：（1）回忆习题 28.2.3 中的 *threatened?*。（2）考虑开发一个抽象函数，用来处理棋盘上所有的元素。这个函数类似于习题 41.2.17 中的 *vec-for-all*。

习题 43.3.2 设计 *unplace-queen*，该函数从棋盘上移去一个皇后以及它所威胁的位置：

```
;; unplace-queen : CB N N -> void
;; 效果：把 CB 中被位于 i 行、j 列的皇后所
;; 威胁的字段设置为 false
(define (unplace-queen CB i j) ...))
```

给定任意的棋盘 *CB*，如下的等式对所有合法的位置 *i* 和 *j* 成立：

---

[1] 在算法中，程序应当为每一个新的状态建立一个全新的棋盘，然后并行地搜索解答。不过，人类会对搜索解答的工作量望而生畏，这也就是为什么人们回避对此进行模拟的原因。

```
(begin
  (place-queen CB i j)
  (unplace-queen CB i j)
  CB)
= CB
```

如果交换两个子表达式，为什么这个等式就不成立了？

习题 43.3.3 使用基于向量的棋盘表示法以及习题 43.3.1 和习题 43.3.2 中的 *place-queen* 和 *unplace-queen*，修改第 28.2 节中皇后问题的解决方法。

习题 43.3.4 使用教学包 draw.ss 开发皇后问题的视图。回忆一下，视图是一个函数，它以图形的方式描述问题的某些方面。这里，自然的解决方法是，依照习题 43.3.3 中的算法显示求解过程的中间阶段，包括回溯步骤。

在第 32.3 节中，我们讨论了单人棋子游戏。这个游戏的目标是一个接一个地移去棋子，直到最后只留下一颗棋子。如果某棵棋子相邻的位置是空的，而且在反方向的位置上有一颗棋子，游戏者就可以移去这个棋子。这时，第二棵棋子跳过第一颗棋子，同时第一颗棋子就被移去。

与皇后问题一样，这个问题的状态也可以用棋子和空位的向量和下标来表示。在真实世界中，移动一颗棋子对应于一个改变棋盘状态的物理动作。在游戏者回溯的时候，这两颗棋子都需要被放回原来的位置。

习题

习题 43.3.5 设计三角形单人棋子游戏棋盘的向量表示法。开发一个函数，用来建立只有一个空位的棋盘。

习题 43.3.6 设计单人棋子游戏中移动的数据表示法。开发执行一次移动的函数。开发回溯一次移动的函数。这两个函数应当是基于专门的效果的。这两个函数满足类似于习题 43.3.2 中 *place-queen* 和 *unplace-queen* 的等式吗？

习题 43.3.7 开发求解单人棋子游戏的回溯算法，游戏中的空位是随机放置的。

习题 43.3.8 使用教学包 draw.ss 开发单人棋子游戏的视图。回忆一下，视图是一个函数，它以图形的方式描述问题的某些方面。这里，自然的解决方法是，依照习题 43.3.7 中的算法显示求解过程的中间阶段，包括回溯的步骤。

结束语

罗森克兰兹：确切来说，你做了什么？

演员：或多或少，我们遵守惯例，考虑所有可能，坚信因果相循。

——汤姆史托帕，《罗森克兰兹和基尔登史坦死了》

不知不觉已到了尾声。尽管还有很多东西要学习，但最好还是先停一停，总结一下，看一看以后还要学习什么。

计　算

从小学到中学，我们学习的是数的计算。首先是对现实存在的东西进行计数，如 3 个苹果、5 个朋友，12 个饼等，当了解了数的含义之后，使用数就不再需要任何实际对象了。

使用软件进行计算遵循的不仅仅是数的代数规律，计算机程序处理的对象包括音乐诗歌、分子细胞、法律案例、电子图表和房屋结构等等。幸运的是，除了数，我们还学习了其他各类信息的表示方法。否则的话，计算和编程将变得索然无味。

首先，计算就是对数据进行合适的操作，有的创建了新值，有的从一个值中提取另一个值，有的修改了值，有的确定一个数据是否属于某一个类型。内部操作和函数也是一种数据，定义对应值的创建，函数应用对应值的提取。

将基本数据操作组合在一起可以定义一个函数。而对于函数组合，有两种机制，一种是将一个函数的值作为另一个函数的参数，另一种则表示若干个函数之间的选择，在函数最终应用时，再激活相应的计算。

在本书，我们学习了基本操作的法则以及将操作组合在一起的法则。有了这些法则，我们就能够理解函数处理输入数据、给出结果并产生效果的原理。比使用纸和笔的你我，计算机更善于运用这些法则，能处理更多的数据、完成更复杂的计算。

程序设计

程序包括定义和表达式。大型程序包括成千上万的定义和表达式，程序设计者除了自己设计函数外，还经常使用他人设计的函数。因此没有强有力的准则，就不能指望开发出高质量的软件。实际上，程序设计是一种刻划计算的方式，也就是使用基本操作的组合对数据进行处理。由此，每一个程序的设计，不管是小型的函数，还是大型的商业软件，都必须从信息上下文环境以及表示信息的数据类型出发，如果问题不寻常或者对我们来说是新的，还必须使用例子来了解问题结构。理解了与项目相关的信息和数据表示之后，就可以作出规划。

项目规划确定了从给定的数据我们希望得到什么结果。在许多情况下，程序对数据进行处理的方式不是一种而是多种。例如，一个管理银行帐户的程序必须处理存款、取款、计算利息、生成税表和其他的任务。在另外的情况下，程序必须计算复杂的关系。例如，一个模拟乒乓球游戏的程序必须计算球的运动、球在球桌上的反弹、球拍对球的作用以及球拍运动，等等。每种情况都需要描述数据处理的方式以及计算之间的关系。接着应该将任务进行排序并从最重要的任务开始设计。可先开发出一个可工作的程序，然后通过增加更多的功能、使其可以处理更多的情况。

设计函数要求对其计算有完全的了解。除非使用精确的语句描述函数的目的和函数的效果，否则是不可能设计出函数的。几乎在所有的情况下，使用例子并且手工计算例子对函数的设计是有帮助的。对于复杂函数或递归函数，还应该使用带目的说明的例子，这对于以后阅读或修改程序的人来说非常有用。研究例子可能揭示基本的设计步骤。在大多数情况下，函数的设计是构造性的，有时也使用累积器或者结构体变化，在个别例子中，还使用了递归。对于这些情况，重要的是解释产生新问题的方法并说明计算什么时候终止。

完成程序的设计之后，还必须对函数进行测试。测试可以发现一些由于各种原因所造成的错误。一个好的测试过程是将那些独立设计的例子转换为测试包，将函数应用于与例子相同的输入，并自动将计算结果和效果与预期的结果和效果进行比较。如果发现不匹配，就给出错误消息。一般来说，测试工作完成之后，测试包也不要删除，而应作为程序的注释，以后若修改程序，还可以修正并使用。

不管多努力，设计函数的过程也不可能在第一次通过测试包时就认为结束了。我们必须考虑函数的设计是否揭示了一些新的有意义的例子，对这些例子应该进行新的测试。有时还必须对程序进行编辑，可能时，适当进行抽象以消除所有的特殊模式。

遵循这些原则，就可以得到结构良好的程序。程序能工作的原因是我们理解了程序做了什么以及是如何工作的。程序中包括的信息可以使那些此后对程序进行修改或增加程序功能的人也能了解程序的工作机理。然而，开发大型软件，还必须按照这些原则不断进行实践，还需要学习更多的程序设计和计算的知识。

继续学习

本书涉及的知识和技巧是进一步学习计算科学、程序设计甚至是进行实际软件开发的基础。第一，掌握 Scheme 语言以及程序设计技术有助于学习当前流行的面向对象程序设计语言，特别是 Java。这两种语言有着共同的思想背景，如计算就是数据处理，程序设计就是描述数据以及对数据进行操作的函数等。与 Scheme 不同的是，Java 要求程序设计者清楚地说明类，并将函数定义放在类的描述之中，它要求程序设计者学习许多语法惯例，因此不适合作为第一种语言。

第二，程序设计者必须学习计算的基本思想。至今为止，我们所学的是面向数据的计算法则。使用本书介绍的程序设计技能，我们可以设计并模拟硬件上的计算。由此，从一个截然不同的侧面观察了计算法则，并产生了下面一些问题：

1. 两种计算机制是截然不同的，那么一个机制是否可以完成另一个机制所完成的计算或者反之？

2. 我们所使用的计算法则是数学的，或者说是抽象的，它没有考虑现实世界的限制，这是否意味着我们能计算任何东西？

3. 硬件计算模拟说明了计算机的能力是有限的，这种限制对我们的计算有何影响？

对这些问题的研究产生了计算机科学，目前仍然是大多数计算机课程的核心内容。

最后，运用本书介绍的程序设计知识，你可以编制许多能解决实际问题的 Scheme 程序，带有内部浏览器和 email 功能的 DrScheme 就是其中一例。然而，开发大型程序还需要学习更多的 Scheme 函数，包括图形用户界面创建、网络互连，事件描述和处理，公共网关接口和 COM 对象等。

所有与 3 个主题相关的材料都可以从本书的站点得到，这些材料拓展了本书的内容，本书官方站点

的地址为：

http://www.htdp.org/

请经常光顾该站点，并继续学习计算和编程知识。

